O'REILLY®

Python

К вершинам мастерства

Лаконичное и эффективное программирование

Второе издание

Лусиану Рамальо

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Лусиану Рамальо

Python – к вершинам мастерства

Лаконичное и эффективное программирование

УДК 004.438Python:004.6
ББК 32.973.22
Р21

Лусиану Рамальо

Р21 Python – к вершинам мастерства: Лаконичное и эффективное программирование / пер. с англ. А. А. Слинкина. 2-е изд. – М.: МК Пресс, 2022. – 898 с.: ил.

ISBN 978-5-97060-885-2

Не тратьте зря времени, пытаясь подогнать Python под способы программирования, знакомые вам по другим языкам. Python настолько прост, что вы очень быстро станете продуктивным программистом, но зачастую это означает, что вы не в полной мере используете то, что может предложить язык. Второе издание книги позволит вам писать более эффективный и современный код на Python 3, обратив себе на пользу лучшие идеи.

Издание предназначено практикующим программистам на Python, которые хотят усовершенствоваться в Python 3.

УДК 004.438Python:004.6
ББК 32.973.22

ISBN (анг.) 978-1-49205-635-5
ISBN (рус.) 978-5-97060-885-2

Марте, с любовью

Оглавление

Предисловие от издательства 19
Отзывы и пожелания 19
Список опечаток 19
Нарушение авторских прав 19

Об авторе 20

Колофон 20

Предисловие 21
На кого рассчитана эта книга 21
На кого эта книга не рассчитана 22
Пять книг в одной 22
Как организована эта книга 22
Практикум 24
Поговорим: мое личное мнение 25
Сопроводительный сайт: fluentpython.com 25
Графические выделения 25
О примерах кода 26
Как с нами связаться 26
Благодарности 27
Благодарности к первому изданию 28

ЧАСТЬ I. СТРУКТУРЫ ДАННЫХ 31

Глава 1. Модель данных в языке Python 32
Что нового в этой главе 33
Колода карт на Python 33
Как используются специальные методы 36
Эмуляция числовых типов 37
Строковое представление 40
Булево значение пользовательского типа 41
API коллекций 41
Сводка специальных методов 43
Почему len – не метод 45
Резюме 45
Дополнительная литература 46

Глава 2. Массив последовательностей 48
Что нового в этой главе 49
Общие сведения о встроенных последовательностях 49
Списковое включение и генераторные выражения 51
Списковое включение и удобочитаемость 52
Сравнение спискового включения с map и filter 53
Декартовы произведения 54
Генераторные выражения 55
Кортеж – не просто неизменяемый список 57
Кортежи как записи 57
Кортежи как неизменяемые списки 58
Сравнение методов кортежа и списка 60
Распаковка последовательностей и итерируемых объектов 61
Распаковка с помощью * в вызовах функций и литеральных последовательностях 63
Распаковка вложенных объектов 63
Сопоставление с последовательностями-образцами 64
Сопоставление с последовательностями-образцами в интерпретаторе 69
Получение среза 72
Почему в срезы и диапазоны не включается последний элемент 73
Объекты среза 73
Многомерные срезы и многоточие 74
Присваивание срезу 75
Использование + и * для последовательностей 76
Построение списка списков 76
Составное присваивание последовательностей 78
Головоломка: присваивание A += 79
Метод list.sort и встроенная функция sorted 81
Когда список не подходит 83
Массивы 83
Представления областей памяти 86
NumPy 88
Двусторонние и другие очереди 90
Резюме 93
Дополнительная литература 94

Глава 3. Словари и множества 99
Что нового в этой главе 99
Современный синтаксис словарей 100
Словарные включения 100
Распаковка отображений 101
Объединение отображений оператором | 102
Сопоставление с отображением-образцом 102
Стандартный API типов отображений 105
Что значит «хешируемый»? 105
Обзор наиболее употребительных методов отображений 106

Вставка и обновление изменяемых значений ... 108
Автоматическая обработка отсутствующих ключей ... 111
defaultdict: еще один подход к обработке отсутствия ключа ... 111
Метод __missing__ ... 112
Несогласованное использование __missing__ в стандартной библиотеке ... 114
Вариации на тему dict ... 115
collections.OrderedDict ... 115
collections.ChainMap ... 116
collections.Counter ... 117
shelve.Shelf ... 117
Создание подкласса UserDict вместо dict ... 118
Неизменяемые отображения ... 120
Представления словаря ... 121
Практические последствия внутреннего устройства класса dict ... 122
Теория множеств ... 123
Литеральные множества ... 125
Множественное включение ... 126
Практические последствия внутреннего устройства класса set ... 126
Операции над множествами ... 127
Теоретико-множественные операции над представлениями словарей ... 129
Резюме ... 131
Дополнительная литература ... 132

Глава 4. Unicode-текст и байты ... 135

Что нового в этой главе ... 136
О символах, и не только ... 136
Все, что нужно знать о байтах ... 137
Базовые кодировщики и декодировщики ... 140
Проблемы кодирования и декодирования ... 141
Обработка UnicodeEncodeError ... 142
Обработка UnicodeDecodeError ... 143
Исключение SyntaxError при загрузке модулей с неожиданной кодировкой ... 144
Как определить кодировку последовательности байтов ... 145
BOM: полезный крокозябр ... 146
Обработка текстовых файлов ... 147
Остерегайтесь кодировок по умолчанию ... 150
Нормализация Unicode для надежного сравнения ... 155
Сворачивание регистра ... 158
Служебные функции для сравнения нормализованного текста ... 158
Экстремальная «нормализация»: удаление диакритических знаков ... 159
Сортировка Unicode-текстов ... 162
Сортировка с помощью алгоритма упорядочивания Unicode ... 164
База данных Unicode ... 165
Поиск символов по имени ... 165
Символы, связанные с числами ... 167

Двухрежимный API ... 168
str и bytes в регулярных выражениях ... 168
str и bytes в функциях из модуля os ... 170
Резюме ... 170
Дополнительная литература ... 171

Глава 5. Построители классов данных ... 176

Что нового в этой главе ... 177
Обзор построителей классов данных ... 177
Основные возможности ... 179
Классические именованные кортежи ... 181
Типизированные именованные кортежи ... 184
Краткое введение в аннотации типов ... 185
Никаких последствий во время выполнения ... 185
Синтаксис аннотаций переменных ... 186
Семантика аннотаций переменных ... 186
Инспекция typing.NamedTuple ... 187
Инспектирование класса с декоратором dataclass ... 188
Еще о @dataclass ... 190
Опции полей ... 191
Постинициализация ... 194
Типизированные атрибуты класса ... 196
Инициализируемые переменные, не являющиеся полями ... 196
Пример использования @dataclass: запись о ресурсе из дублинского ядра ... 197
Класс данных как признак кода с душком ... 199
Класс данных как временная конструкция ... 201
Класс данных как промежуточное представление ... 201
Сопоставление с экземплярами классов – образцами ... 201
Простые классы-образцы ... 202
Именованные классы-образцы ... 202
Позиционные классы-образцы ... 204
Резюме ... 205
Дополнительная литература ... 205

Глава 6. Ссылки на объекты, изменяемость и повторное использование ... 209

Что нового в этой главе ... 210
Переменные – не ящики ... 210
Тождественность, равенство и псевдонимы ... 212
Выбор между == и is ... 213
Относительная неизменяемость кортежей ... 214
По умолчанию копирование поверхностное ... 215
Глубокое и поверхностное копирование произвольных объектов ... 218
Параметры функций как ссылки ... 219
Значения по умолчанию изменяемого типа: неудачная мысль ... 220
Защитное программирование при наличии изменяемых параметров ... 222

del и сборка мусора....224
Как Python хитрит с неизменяемыми объектами....226
Резюме....228
Дополнительная литература....229

ЧАСТЬ II. ФУНКЦИИ КАК ОБЪЕКТЫ....233

Глава 7. Функции как полноправные объекты....234

Что нового в этой главе....235
Обращение с функцией как с объектом....235
Функции высшего порядка....236
Современные альтернативы функциям map, filter и reduce....237
Анонимные функции....239
Девять видов вызываемых объектов....240
Пользовательские вызываемые типы....241
От позиционных к чисто именованным параметрам....242
Чисто позиционные параметры....244
Пакеты для функционального программирования....245
Модуль operator....245
Фиксация аргументов с помощью functools.partial....248
Резюме....250
Дополнительная литература....250

Глава 8. Аннотации типов в функциях....254

Что нового в этой главе....255
О постепенной типизации....255
Постепенная типизация на практике....256
Начинаем работать с Муру....257
А теперь построже....258
Значение параметра по умолчанию....258
None в качестве значения по умолчанию....260
Типы определяются тем, какие операции они поддерживают....261
Типы, пригодные для использования в аннотациях....266
Тип Any....266
«Является подтипом» и «совместим с»....267
Простые типы и классы....269
Типы Optional и Union....269
Обобщенные коллекции....270
Типы кортежей....273
Обобщенные отображения....275
Абстрактные базовые классы....276
Тип Iterable....278
Параметризованные обобщенные типы и TypeVar....280
Статические протоколы....284
Тип Callable....288
Тип NoReturn....291
Аннотирование чисто позиционных и вариадических параметров....291

Несовершенная типизация и строгое тестирование ... 292
Резюме ... 293
Дополнительная литература ... 294

Глава 9. Декораторы и замыкания ... 300

Что нового в этой главе ... 301
Краткое введение в декораторы ... 301
Когда Python выполняет декораторы ... 302
Регистрационные декораторы ... 304
Правила видимости переменных ... 304
Замыкания ... 307
Объявление nonlocal ... 310
Логика поиска переменных ... 311
Реализация простого декоратора ... 312
Как это работает ... 313
Декораторы в стандартной библиотеке ... 314
Запоминание с помощью functools.cache ... 315
Использование lru_cache ... 317
Обобщенные функции с одиночной диспетчеризацией ... 318
Параметризованные декораторы ... 322
Параметризованный регистрационный декоратор ... 323
Параметризованный декоратор clock ... 324
Декоратор clock на основе класса ... 327
Резюме ... 328
Дополнительная литература ... 328

Глава 10. Реализация паттернов проектирования с помощью полноправных функций ... 333

Что нового в этой главе ... 334
Практический пример: переработка паттерна Стратегия ... 334
Классическая Стратегия ... 334
Функционально-ориентированная стратегия ... 338
Выбор наилучшей стратегии: простой подход ... 341
Поиск стратегий в модуле ... 342
Паттерн Стратегия, дополненный декоратором ... 343
Паттерн Команда ... 345
Резюме ... 346
Дополнительная литература ... 347

ЧАСТЬ III. КЛАССЫ И ПРОТОКОЛЫ ... 351

Глава 11. Объект в духе Python ... 352

Что нового в этой главе ... 353
Представления объекта ... 353
И снова класс вектора ... 354
Альтернативный конструктор ... 356
Декораторы classmethod и staticmethod ... 357

Форматирование при выводе 358
Хешируемый класс Vector2d 361
Поддержка позиционного сопоставления с образцом 363
Полный код класса Vector2d, версия 3 365
Закрытые и «защищенные» атрибуты в Python 368
Экономия памяти с помощью атрибута класса __slots__ 370
Простое измерение экономии, достигаемой за счет __slot__ 372
Проблемы при использовании __slots__ 373
Переопределение атрибутов класса 374
Резюме 376
Дополнительная литература 377

Глава 12. Специальные методы для последовательностей 381

Что нового в этой главе 381
Vector: пользовательский тип последовательности 382
Vector, попытка № 1: совместимость с Vector2d 382
Протоколы и утиная типизация 385
Vector, попытка № 2: последовательность, допускающая срез 386
Как работает срезка 387
Метод __getitem__ с учетом срезов 388
Vector, попытка № 3: доступ к динамическим атрибутам 390
Vector, попытка № 4: хеширование и ускорение оператора == 393
Vector, попытка № 5: форматирование 399
Резюме 406
Дополнительная литература 407

Глава 13. Интерфейсы, протоколы и ABC 411

Карта типизации 412
Что нового в этой главе 413
Два вида протоколов 413
Программирование уток 415
Python в поисках следов последовательностей 415
Партизанское латание как средство реализации протокола во время выполнения 417
Защитное программирование и принцип быстрого отказа 419
Гусиная типизация 421
Создание подкласса ABC 426
ABC в стандартной библиотеке 427
Определение и использование ABC 430
Синтаксические детали ABC 435
Создание подклассов ABC 435
Виртуальный подкласс Tombola 438
Использование функции register на практике 440
ABC и структурная типизация 440
Статические протоколы 442
Типизированная функция double 443
Статические протоколы, допускающие проверку во время выполнения 444

Ограничения протоколов, допускающих проверку во время выполнения ... 447
Поддержка статического протокола ... 448
Проектирование статического протокола ... 450
Рекомендации по проектированию протоколов ... 451
Расширение протокола ... 452
ABC из пакета numbers и числовые протоколы ... 453
Резюме ... 456
Дополнительная литература ... 457

Глава 14. Наследование: к добру или к худу ... 462

Что нового в этой главе ... 463
Функция super() ... 463
Сложности наследования встроенным типам ... 465
Множественное наследование и порядок разрешения методов ... 468
Классы-примеси ... 473
Отображения, не зависящие от регистра ... 473
Множественное наследование в реальном мире ... 475
ABC – тоже примеси ... 475
ThreadingMixIn и ForkingMixIn ... 475
Множественное наследование в Tkinter ... 480
Жизнь с множественным наследованием ... 482
Предпочитайте композицию наследованию класса ... 483
Разберитесь, зачем наследование используется в каждом конкретном случае ... 483
Определяйте интерфейсы явно с помощью ABC ... 483
Используйте примеси для повторного использования кода ... 484
Предоставляйте пользователям агрегатные классы ... 484
Наследуйте только классам, предназначенным для наследования ... 484
Воздерживайтесь от наследования конкретным классам ... 485
Tkinter: хороший, плохой, злой ... 485
Резюме ... 487
Дополнительная литература ... 488

Глава 15. Еще об аннотациях типов ... 492

Что нового в этой главе ... 492
Перегруженные сигнатуры ... 492
Перегрузка max ... 494
Уроки перегрузки max ... 498
TypedDict ... 498
Приведение типов ... 505
Чтение аннотаций типов во время выполнения ... 508
Проблемы с аннотациями во время выполнения ... 508
Как решать проблему ... 511
Реализация обобщенного класса ... 511
Основы терминологии, относящейся к обобщенным типам ... 513
Вариантность ... 514

Инвариантный разливочный автомат ... 514
Ковариантный разливочный автомат ... 516
Контравариантная урна ... 516
Обзор вариантности ... 518
Реализация обобщенного статического протокола ... 520
Резюме ... 522
Дополнительная литература ... 523

Глава 16. Перегрузка операторов ... 528

Что нового в этой главе ... 529
Основы перегрузки операторов ... 529
Унарные операторы ... 530
Перегрузка оператора сложения векторов + ... 533
Перегрузка оператора умножения на скаляр * ... 538
Использование @ как инфиксного оператора ... 540
Арифметические операторы – итоги ... 541
Операторы сравнения ... 542
Операторы составного присваивания ... 545
Резюме ... 549
Дополнительная литература ... 550

ЧАСТЬ IV. ПОТОК УПРАВЛЕНИЯ ... 555

Глава 17. Итераторы, генераторы и классические сопрограммы ... 556

Что нового в этой главе ... 557
Последовательность слов ... 557
Почему последовательности итерируемы: функция iter ... 558
Использование iter в сочетании с Callable ... 560
Итерируемые объекты и итераторы ... 561
Классы Sentence с методом __iter__ ... 564
Класс Sentence, попытка № 2: классический итератор ... 565
Не делайте итерируемый объект итератором для самого себя ... 566
Класс Sentence, попытка № 3: генераторная функция ... 567
Как работает генератор ... 568
Ленивые классы Sentence ... 570
Класс Sentence, попытка № 4: ленивый генератор ... 570
Класс Sentence, попытка № 5: генераторное выражение ... 571
Генераторные выражения: когда использовать ... 573
Генератор арифметической прогрессии ... 575
Построение арифметической прогрессии с помощью itertools ... 577
Генераторные функции в стандартной библиотеке ... 578
Функции редуцирования итерируемого объекта ... 588
yield from и субгенераторы ... 590
Изобретаем chain заново ... 591
Обход дерева ... 592
Обобщенные итерируемые типы ... 596

Классические сопрограммы ... 597
Пример: сопрограмма для вычисления накопительного среднего ... 599
Возврат значения из сопрограммы ... 601
Аннотации обобщенных типов для классических сопрограмм ... 605
Резюме ... 607
Дополнительная литература ... 607

Глава 18. Блоки with, match и else ... 612

Что нового в этой главе ... 613
Контекстные менеджеры и блоки with ... 613
Утилиты contextlib ... 617
Использование @contextmanager ... 618
Сопоставление с образцом в lis.py: развернутый пример ... 622
Синтаксис Scheme ... 622
Предложения импорта и типы ... 623
Синтаксический анализатор ... 624
Класс Environment ... 626
Цикл REPL ... 628
Вычислитель ... 629
Procedure: класс, реализующий замыкание ... 636
Использование OR-образцов ... 637
Делай то, потом это: блоки else вне if ... 638
Резюме ... 640
Дополнительная литература ... 641

Глава 19. Модели конкурентности в Python ... 646

Что нового в этой главе ... 647
Общая картина ... 647
Немного терминологии ... 648
Процессы, потоки и знаменитая блокировка GIL в Python ... 650
Конкурентная программа Hello World ... 652
Анимированный индикатор с потоками ... 652
Индикатор с процессами ... 655
Индикатор с сопрограммами ... 656
Сравнение супервизоров ... 660
Истинное влияние GIL ... 662
Проверка знаний ... 662
Доморощенный пул процессов ... 665
Решение на основе процессов ... 666
Интерпретация времени работы ... 667
Код проверки на простоту для многоядерной машины ... 668
Эксперименты с большим и меньшим числом процессов ... 671
Не решение на основе потоков ... 672
Python в многоядерном мире ... 673
Системное администрирование ... 674
Наука о данных ... 675
Веб-разработка на стороне сервера и на мобильных устройствах ... 676

WSGI-серверы приложений ... 678
Распределенные очереди задач ... 680
Резюме ... 681
Дополнительная литература ... 682
Конкурентность с применением потоков и процессов ... 682
GIL ... 684
Конкурентность за пределами стандартной библиотеки ... 684
Конкурентность и масштабируемость за пределами Python ... 686

Глава 20. Конкурентные исполнители ... 691

Что нового в этой главе ... 691
Конкурентная загрузка из веба ... 692
Скрипт последовательной загрузки ... 694
Загрузка с применением библиотеки concurrent.futures ... 696
Где находятся будущие объекты? ... 698
Запуск процессов с помощью concurrent.futures ... 701
И снова о проверке на простоту на многоядерной машине ... 701
Эксперименты с Executor.map ... 704
Загрузка с индикацией хода выполнения и обработкой ошибок ... 707
Обработка ошибок во flags2-примерах ... 711
Использование futures.as_completed ... 713
Резюме ... 716
Дополнительная литература ... 716

Глава 21. Асинхронное программирование ... 719

Что нового в этой главе ... 720
Несколько определений ... 720
Пример использования asyncio: проверка доменных имен ... 721
Предложенный Гвидо способ чтения асинхронного кода ... 723
Новая концепция: объекты, допускающие ожидание ... 724
Загрузка файлов с помощью asyncio и HTTPX ... 725
Секрет платформенных сопрограмм: скромные генераторы ... 727
Проблема «все или ничего» ... 728
Асинхронные контекстные менеджеры ... 729
Улучшение асинхронного загрузчика ... 730
Использование asyncio.as_completed и потока ... 731
Регулирование темпа запросов с помощью семафора ... 733
Отправка нескольких запросов при каждой загрузке ... 736
Делегирование задач исполнителям ... 739
Написание асинхронных серверов ... 740
Веб-служба FastAPI ... 742
Асинхронный TCP-сервер ... 746
Асинхронные итераторы и итерируемые объекты ... 751
Асинхронные генераторные функции ... 752
Асинхронные включения и асинхронные генераторные выражения ... 758
async за пределами asyncio: Curio ... 760
Аннотации типов для асинхронных объектов ... 763

Как работает и как не работает асинхронность 764
Круги, разбегающиеся вокруг блокирующих вызовов 764
Миф о системах, ограниченных вводом-выводом 765
Как не попасть в ловушку счетных функций 765
Резюме 766
Дополнительная литература 767

ЧАСТЬ V. МЕТАПРОГРАММИРОВАНИЕ 771

Глава 22. Динамические атрибуты и свойства 772

Что нового в этой главе 772
Применение динамических атрибутов для обработки данных 773
Исследование JSON-подобных данных с динамическими атрибутами 775
Проблема недопустимого имени атрибута 778
Гибкое создание объектов с помощью метода __new__ 779
Вычисляемые свойства 781
Шаг 1: создание управляемого данными атрибута 782
Шаг 2: выборка связанных записей с помощью свойств 784
Шаг 3: переопределение существующего атрибута свойством 787
Шаг 4: кеширование свойств на заказ 788
Шаг 5: кеширование свойств с помощью functools 789
Использование свойств для контроля атрибутов 791
LineItem, попытка № 1: класс строки заказа 791
LineItem, попытка № 2: контролирующее свойство 792
Правильный взгляд на свойства 794
Свойства переопределяют атрибуты экземпляра 795
Документирование свойств 797
Программирование фабрики свойств 798
Удаление атрибутов 800
Важные атрибуты и функции для работы с атрибутами 802
Специальные атрибуты, влияющие на обработку атрибутов 802
Встроенные функции для работы с атрибутами 803
Специальные методы для работы с атрибутами 804
Резюме 805
Дополнительная литература 806

Глава 23. Дескрипторы атрибутов 810

Что нового в этой главе 810
Пример дескриптора: проверка значений атрибутов 811
LineItem попытка № 3: простой дескриптор 811
LineItem попытка № 4: автоматическое генерирование имен атрибутов хранения 816
LineItem попытка № 5: новый тип дескриптора 818
Переопределяющие и непереопределяющие дескрипторы 820
Переопределяющие дескрипторы 822
Переопределяющий дескриптор без __get__ 823
Непереопределяющий дескриптор 824
Перезаписывание дескриптора в классе 825

Методы являются дескрипторами 826
Советы по использованию дескрипторов 828
Строка документации дескриптора и перехват удаления 829
Резюме 831
Дополнительная литература 831

Глава 24. Метапрограммирование классов 834

Что нового в этой главе 835
Классы как объекты 835
type: встроенная фабрика классов 836
Функция-фабрика классов 837
Введение в __init_subclass__ 840
Почему __init_subclass__ не может конфигурировать __slots__ 846
Дополнение класса с помощью декоратора класса 847
Что когда происходит: этап импорта и этап выполнения 849
Демонстрация работы интерпретатора 850
Основы метаклассов 854
Как метакласс настраивает класс 856
Элегантный пример метакласса 857
Демонстрация работы метакласса 860
Реализация Checked с помощью метакласса 864
Метаклассы на практике 868
Современные средства позволяют упростить или заменить метаклассы 868
Метаклассы – стабильное языковое средство 869
У класса может быть только один метакласс 869
Метаклассы должны быть деталью реализации 870
Метаклассный трюк с __prepare__ 870
Заключение 872
Резюме 873
Дополнительная литература 874

Послесловие 878

Предметный указатель 881

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу *dmkpress@gmail.com*, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Лусиану Рамальо был веб-разработчиком до выхода компании Netscape на IPO в 1995 году, а в 1998 году перешел с Perl на Java, а затем на Python. В 2015 году пришел в компанию Thoughtworks, где работает главным консультантом в отделении в Сан-Паулу. Он выступал с основными докладами, презентациями и пособиями на различных мероприятиях, связанных с Python, в обеих Америках, Европе и Азии. Выступал также на конференциях по Go и Elixir по вопросам проектирования языков. Рамальо – член фонда Python Software Foundation и сооснователь клуба Garoa Hacker Clube, первого места для общения хакеров в Бразилии.

Колофон

На обложке изображена намакская песчаная ящерица (*Pedioplanis namaquensis*), встречающаяся в засушливых саваннах и полупустынях Намибии.

Внешние признаки: туловище черное с четырьмя белыми полосками на спине, лапы коричневые с белыми пятнышками, брюшко белое, длинный розовато-коричневый хвост. Одна из самых быстрых ящериц, активна в течение дня, питается мелкими насекомыми. Обитает на бедных растительностью песчано-каменистых равнинах. Самка откладывает от трех до пяти яиц в ноябре. Остаток зимы ящерицы спят в норах, которые роют в корнях кустов.

В настоящее время охранный статус намакской песчаной ящерицы – «пониженная уязвимость». Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой вымирания; все они важны для мира.

Предисловие

> План такой: если кто-то пользуется средством, которое вы не понимаете, просто пристрелите его. Это проще, чем учить что-то новое, и очень скоро в мире останутся только кодировщики, которые используют только всем понятное крохотное подмножество Python 0.9.6 <смешок>.
>
> – Тим Питерс, легендарный разработчик ядра и автор сборника поучений «The Zen of Python»[1]

«Python – простой для изучения и мощный язык программирования». Это первые слова в официальном «Пособии по Python» (*https://docs.python.org/3/tutorial/*). И это правда, но не вся правда: поскольку язык так просто выучить и начать применять на деле, многие практикующие программисты используют лишь малую часть его обширных возможностей.

Опытный программист может написать полезный код на Python уже через несколько часов изучения. Но вот проходят недели, месяцы – и многие разработчики так и продолжают писать на Python код, в котором отчетливо видно влияние языков, которые они учили раньше. И даже если Python – ваш первый язык, все равно авторы академических и вводных учебников зачастую излагают его, тщательно избегая особенностей, характерных только для этого языка.

Будучи преподавателем, который знакомит с Python программистов, знающих другие языки, я нередко сталкиваюсь еще с одной проблемой, которую пытаюсь решить в этой книге: нас интересует только то, о чем мы уже знаем. Любой программист, знакомый с каким-то другим языком, догадывается, что Python поддерживает регулярные выражения, и начинает смотреть, что про них написано в документации. Но если вы никогда раньше не слыхали о распаковке кортежей или о дескрипторах, то, скорее всего, и искать сведения о них не станете, а в результате не будете использовать эти средства лишь потому, что они специфичны для Python.

Эта книга не является полным справочным руководством по Python. Упор в ней сделан на языковые средства, которые либо уникальны для Python, либо отсутствуют во многих других популярных языках. Кроме того, в книге рассматривается в основном ядро языка и немногие библиотеки. Я редко упоминаю о пакетах, не включенных в стандартную библиотеку, хотя нынче количество пакетов для Python уже перевалило за 60 000 и многие из них исключительно полезны.

На кого рассчитана эта книга

Эта книга написана для практикующих программистов на Python, которые хотят усовершенствоваться в Python 3. Я тестировал примеры на Python 3.10,

[1] Сообщение в группе Usenet comp.lang.python от 23 декабря 2002: «Acrimony in c.l.p.» (*https://mail.python.org/pipermail/python-list/2002-December/147293.html*).

а большую их часть также на Python 3.9 и 3.8. Если какой-то пример требует версии 3.10, то это явно оговаривается.

Если вы не уверены в том, достаточно ли хорошо знаете Python, чтобы читать эту книгу, загляните в оглавление официального «Пособия по Python» (*https://docs.python.org/3/tutorial/*). Темы, рассмотренные в пособии, в этой книге не затрагиваются, за исключением некоторых новых средств.

На кого эта книга не рассчитана

Если вы только начинаете изучать Python, эта книга покажется вам сложноватой. Более того, если вы откроете ее на слишком раннем этапе путешествия в мир Python, то может сложиться впечатление, будто в каждом Python-скрипте следует использовать специальные методы и приемы метапрограммирования. Преждевременное абстрагирование ничем не лучше преждевременной оптимизации.

Пять книг в одной

Я рекомендую всем прочитать главу 1 «Модель данных в языке Python». Читатели этой книги в большинстве своем после ознакомления с главой 1, скорее всего, смогут легко перейти к любой части, но я зачастую предполагаю, что главы каждой части читаются по порядку. Части I–V можно рассматривать как отдельные книги внутри книги.

Я старался сначала рассказывать о том, что уже есть, а лишь затем о том, как создавать что-то свое. Например, в главе 2 части II рассматриваются готовые типы последовательностей, в том числе не слишком хорошо известные, например `collections.deque`. О создании пользовательских последовательностей речь пойдет только в части III, где мы также узнаем об использовании абстрактных базовых классов (abstract base classes – ABC) из модуля `collections.abc`. Создание собственного ABC обсуждается еще позже, поскольку я считаю, что сначала нужно освоиться с использованием ABC, а уж потом писать свои.

У такого подхода несколько достоинств. Прежде всего, зная, что есть в вашем распоряжении, вы не станете заново изобретать велосипед. Мы пользуемся готовыми классами коллекций чаще, чем реализуем собственные, и можем уделить больше внимания нетривиальным способам работы с имеющимися средствами, отложив на потом разговор о разработке новых. И мы скорее унаследуем существующему абстрактному базовому классу, чем будем создавать новый с нуля. Наконец, я полагаю, что понять абстракцию проще после того, как видел ее в действии.

Недостаток же такой стратегии в том, что главы изобилуют ссылками на более поздние материалы. Надеюсь, что теперь, когда вы узнали, почему я избрал такой путь, вам будет проще смириться с этим.

Как организована эта книга

Ниже описаны основные темы, рассматриваемые в каждой части книги.

Часть I «Структуры данных»

В главе I, посвященной модели данных в Python, объясняется ключевая роль специальных методов (например, `__repr__`) для обеспечения единообразного поведения объектов любого типа. Специальные методы более подробно обсуждаются на протяжении всей книги. В остальных главах этой части рассматривается использование типов коллекций: последовательностей, отображений и множеств, а также различие между типами `str` и `bytes` – то, что радостно приветствовали пользователи Python 3 и чего отчаянно не хватает пользователям Python 2, еще не модернизировавшим свой код. Также рассматриваются высокоуровневые построители классов, имеющиеся в стандартной библиотеке: фабрики именованных кортежей и декоратор `@dataclass`. Сопоставление с образцом – новая возможность, появившаяся в Python 3.10, – рассматривается в разделах глав 2, 3 и 5, где обсуждаются паттерны последовательностей, отображений и классов. Последняя глава части I посвящена жизненному циклу объектов: ссылкам, изменяемости и сборке мусора.

Часть II «Функции как объекты»

Здесь речь пойдет о функциях как полноправных объектах языка: что под этим понимается, как это отражается на некоторых популярных паттернах проектирования и как реализовать декораторы функций с помощью замыканий. Рассматриваются также следующие вопросы: общая идея вызываемых объектов, атрибуты функций, интроспекция, аннотации параметров и появившееся в Python 3 объявление `nonlocal`. Глава 8 содержит введение в новую важную тему – аннотации типов в сигнатурах функций.

Часть III «Классы и протоколы»

Теперь наше внимание перемещается на создание классов «вручную» – в отличие от использования построителей классов, рассмотренных в главе 5. Как и в любом объектно-ориентированном (ОО) языке, в Python имеется свой набор средств; какие-то из них, возможно, присутствовали в языке, с которого вы и я начинали изучение программирования на основе классов, а какие-то – нет. В главах из этой части объясняется, как создать свою коллекцию, абстрактный базовый класс (ABC) и протокол, как работать со множественным наследованием и как реализовать перегрузку операторов (если это имеет смысл). В главе 15 мы продолжим обсуждать аннотации типов.

Часть IV «Поток управления»

Эта часть посвящена языковым конструкциям и библиотекам, выходящим за рамки последовательного потока управления с его условными выражениями, циклами и подпрограммами. Сначала мы рассматриваем генераторы, затем – контекстные менеджеры и сопрограммы, в том числе трудную для понимания, но исключительно полезную новую конструкцию `yield from`. В главу 18 включен важный пример использования сопоставления с образцом в простом, но функциональном интерпретаторе языка. Глава 19 «Модели конкурентности в Python» новая, она посвящена обзору различных

видов конкурентной и параллельной обработки в Python, их ограничений и вопросу о том, как архитектура программы позволяет использовать Python в приложениях масштаба веба. Я переписал главу об *асинхронном программировании*, стремясь уделить больше внимания базовым средствам языка – `await`, `async dev`, `async for` и `async with` – и показать, как они используются совместно с библиотекой *asyncio* и другими каркасами.

Часть V «Метапрограммирование»

Эта часть начинается с обзора способов построения классов с динамически создаваемыми атрибутами для обработки слабоструктурированных данных, например в формате JSON. Затем мы рассматриваем знакомый механизм свойств, после чего переходим к низкоуровневым деталям доступа к атрибутам объекта с помощью дескрипторов. Объясняется связь между функциями, методами и дескрипторами. На примере приведенной здесь пошаговой реализации библиотеки контроля полей мы вскрываем тонкие нюансы, которые делают необходимым применение рассмотренных в этой главе продвинутых инструментов: декораторов классов и метаклассов.

Практикум

Часто для исследования языка и библиотек мы будем пользоваться интерактивной оболочкой Python. Я считаю важным всячески подчеркивать удобство этого средства для обучения. Особенно это относится к читателям, привыкшим к статическим компилируемым языкам, в которых нет цикла чтения-вычисления-печати (read-eval-print#loop – REPL).

Один из стандартных пакетов тестирования для Python, `doctest` (*https://docs.python.org/3/library/doctest.html*), работает следующим образом: имитирует сеансы оболочки и проверяет, что результат вычисления выражения совпадает с заданным. Я использовал `doctest` для проверки большей части приведенного в книге кода, включая листинги сеансов оболочки. Для чтения книги ни применять, ни даже знать о пакете `doctest` не обязательно: основная характеристика doctest-скриптов (или просто тестов) состоит в том, что они выглядят как копии интерактивных сеансов оболочки Python, поэтому вы можете сами выполнить весь демонстрационный код.

Иногда я буду объяснять, чего мы хотим добиться, демонстрируя тест раньше кода, который заставляет его выполниться успешно. Если сначала отчетливо представить себе, что необходимо сделать, а только потом задумываться о том, как это сделать, то структура кода заметно улучшится. Написание тестов раньше кода – основа методологии разработки через тестирование (TDD); мне кажется, что и для преподавания это полезно. Если вы незнакомы с `doctest`, загляните в документацию (*https://docs.python.org/3/library/doctest.html*) и в репозиторий исходного кода к этой книге (*https://github.com/fluentpython/example-code-2e*).

Я также написал автономные тесты для нескольких крупных примеров, воспользовавшись модулем `pytest`, который, как мне кажется, проще и имеет больше возможностей, чем модуль `unittest` из стандартной библиотеки. Вы увидите, что для проверки правильности большей части кода в книге до-

статочно ввести команду `python3 -m doctest example_script.py` или `pytest` в оболочке ОС. Конфигурационный файл *pytest.ini* в корне репозитория исходного кода (*https://github.com/fluentpython/example-code-2e*) гарантирует, что команда `pytest` найдет и выполнит тесты.

Поговорим: мое личное мнение

Я использую, преподаю и принимаю участие в обсуждениях Python с 1998 года и обожаю изучать и сравнивать разные языки программирования, их дизайн и теоретические основания. В конце некоторых глав имеются врезки «Поговорим», где излагается моя личная точка зрения на Python и другие языки. Если вас такие обсуждения не интересуют, можете спокойно пропускать их. Приведенные в них сведения всегда факультативны.

Сопроводительный сайт: fluentpython.com

Из-за включения новых средств – аннотаций типов, классов данных, сопоставления с образцом и других – это издание оказалось почти на 30 % больше первого. Чтобы книга была подъемной, я перенес часть материалов на сайт *fluentpython.com*. В нескольких главах имеются ссылки на опубликованные там статьи. Там же есть текст нескольких выборочных глав. Полный текст доступен онлайн (*https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348/*) и по подписке O'Reilly Learning *(https://www.oreilly.com/online-learning/try-now.html*). Код примеров имеется в репозитории на GitHub (*https://github.com/fluentpython/example-code-2e*).

Графические выделения

В книге применяются следующие графические выделения.

Курсив

Новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширинный`

Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

Отмечу, что когда внутри элемента, набранного моноширинным шрифтом, оказывается разрыв строки, дефис не добавляется, поскольку он мог бы быть ошибочно принят за часть элемента.

`Моноширинный полужирный`

Команды или иной текст, который пользователь должен вводить буквально.

`Моноширинный курсив`

Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.

Так обозначается совет или рекомендация.

Так обозначается замечание общего характера.

Так обозначается предупреждение или предостережение.

О примерах кода

Все скрипты и большая часть приведенных в книге фрагментов кода имеются в репозитории на GitHub по адресу (*https://github.com/fluentpython/example-code-2e*).

Вопросы технического характера, а также замечания по примерам кода следует отправлять по адресу *bookquestions@oreilly.com*.

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешения необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Fluent Python, 2nd ed., by Luciano Ramalho (O'Reilly). Copyright 2022 Luciano Ramalho, 978-1-492-05635-5».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)

707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: *https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348/*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *http://www.oreilly.com*.

Читайте нас на Facebook: *http://facebook.com/oreilly*.

Следите за нашей лентой в Twitter: *http://twitter.com/oreillymedia*.

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

Благодарности

Я не думал, что подготовка нового издания книги по Python спустя пять лет после выхода первого станет таким серьезным предприятием, каким оно оказалось. Марта Мелло, моя любимая супруга, всегда была рядом, когда я в ней нуждался. Мой дорогой друг Леонардо Рохаэль помогал мне с самого начала и до последней технической рецензии, включая сведение и перепроверку отзывов других рецензентов, читателей и редакторов. Честно – не знаю, справился бы я без вашей поддержки, Марта и Лео. От всей души спасибо!

Юрген Гмах (Jürgen Gmach), Калеб Хэттинг (Caleb Hattingh), Джесс Мейлз (Jess Males), Леонардо Рохаэль (Leonardo Rochael) и Мирослав Седивы (Miroslav Šedivý) составили выдающуюся команду технических рецензентов для второго издания. Они просмотрели всю книгу. Билл Берман (Bill Behrman), Брюс Эккель (Bruce Eckel), Ренато Оливейра (Renato Oliveira) и Родриго Бернардо Пиментель (Rodrigo Bernardo Pimentel) отрецензировали отдельные главы. Многие их предложения позволили значительно улучшить книгу.

Многие читатели присылали исправления и по-другому вносили свой вклад, пока книга готовилась к выходу: Guilherme Alves, Christiano Anderson, Konstantin Baikov, K. Alex Birch, Michael Boesl, Lucas Brunialti, Sergio Cortez, Gino Crecco, Chukwuerika Dike, Juan Esteras, Federico Fissore, Will Frey, Tim Gates, Alexander Hagerman, Chen Hanxiao, Sam Hyeong, Simon Ilincev, Parag Kalra, Tim King, David Kwast, Tina Lapine, Wanpeng Li, Guto Maia, Scott Martindale, Mark Meyer, Andy McFarland, Chad McIntire, Diego Rabatone Oliveira, Francesco Piccoli, Meredith Rawls, Michael Robinson, Federico Tula Rovaletti, Tushar Sadhwani, Arthur Constantino Scardua, Randal L. Schwartz, Avichai Sefati, Guannan Shen, William Simpson, Vivek Vashist, Jerry Zhang, Paul Zuradzki и другие, пожелавшие остаться неназванными или неупомянутые, потому что я вовремя не записал их имена, за что прощу прощения.

По ходу работы я много узнал о типах, конкурентности, сопоставлении с образцом и метапрограммировании, общаясь с Майклом Альбертом (Michael Albert), Пабло Агиларом (Pablo Aguilar), Калебом Барретом (Kaleb Barrett), Дэвидом Бизли (David Beazley), Х. С. О. Буэно (J. S. O. Bueno), Брюсом Эккелем (Bruce Eckel), Мартином Фаулером (Martin Fowler), Иваном Левкивским (Ivan Levkivskyi), Алексом Мартелли (Alex Martelli), Питером Норвигом (Peter Norvig),

Себастьяном Риттау (Sebastian Rittau), Гвидо ван Россумом (Guido van Rossum), Кэрол Уиллинг (Carol Willing) и Джелле Зийлстра (Jelle Zijlstra).

Редакторы O'Reilly Джефф Блейел (Jeff Bleiel), Джилл Леонард (Jill Leonard) и Амелия Блевинс (Amelia Blevins) предложили, как можно улучшить порядок изложения во многих местах книги. Джефф Блейел и выпускающий редактор Дэнни Элфанбаум (Danny Elfanbaum) оказывали мне поддержку на протяжении всего долгого марафона.

Идеи и предложения каждого из них сделали книгу лучше и точнее. Но какие-то ошибки в конечном продукте неизбежно остались – всю ответственность за них несу я, заранее приношу извинения.

Наконец, я хочу сердечно поблагодарить всех своих коллег по компании Thoughtworks Brazil, а особенно моего спонсора Алексея Боаша (Alexey Bôas), который постоянно поддерживал меня самыми разными способами.

Конечно же, все, кто помогал мне разобраться в Python и написать первое издание этой книги, заслуживают благодарности вдвойне. Если бы первое издание не пользовалось успехом, то второго бы просто не было.

Благодарности к первому изданию

Комплект шахматных фигур в стиле Баухаус, изготовленный Йозефом Хартвигом, – пример великолепного дизайна: просто, красиво и понятно. Гвидо ван Россум, сын архитектора и брат дизайнера базовых шрифтов, создал шедевр дизайна языков программирования. Мне нравится преподавать Python, потому что это красивый, простой и понятный язык.

Алекс Мартелли (Alex Martelli) и Анна Равенскрофт (Anna Ravenscroft) первыми познакомились с черновиком книги и рекомендовали предложить ее издательству O'Reilly для публикации. Написанные ими книги научили меня идиоматике языка Python и явили образцы ясности, точности и глубины технического текста. Свыше 5000 сообщений, написанных Алексом на сайте Stack Overflow *(http://stackoverflow.com/users/95810/alexmartelli)*, – неиссякаемый источник глубоких мыслей о языке и его правильном использовании.

Мартелли и Равенскрофт были также техническими рецензентами книги наряду с Леннартом Регебро (Lennart Regebro) и Леонардо Рохаэлем (Leonardo Rochael). У каждого члена этой выдающейся команды рецензентов за плечами не менее 15 лет работы с Python и многочисленные вклады в популярные проекты. Все они находятся в тесном контакте с другими разработчиками из сообщества. Я получил от них сотни исправлений, предложений, вопросов и отзывов, благодаря чему книга стала значительно лучше. Виктор Стиннер (Victor Stinner) любезно отрецензировал главу 18, привнеся в команду свой опыт сопровождения модуля `asyncio`. Работать вместе с ними на протяжении нескольких месяцев стало для меня высокой честью и огромным удовольствием.

Редактор Меган Бланшетт (Meghan Blanchette) оказалась великолепным наставником и помогла мне улучшить структуру книги. Она подсказывала, когда книгу становилось скучно читать, и не давала топтаться на месте. Брайан Макдональд (Brian MacDonald) редактировал главы из части III во время отсутствия Меган. Мне очень понравилось работать с ними, да и вообще со всеми сотрудниками издательства O'Reilly, включая команду разработки и поддерж-

ки Atlas (Atlas – это платформа книгоиздания O'Reilly, на которой мне посчастливилось работать).

Марио Доменик Гулар (Mario Domenech Goulart) вносил многочисленные предложения, начиная с самого первого варианта книги для предварительного ознакомления. Я также получал ценные отзывы от Дэйва Поусона (Dave Pawson), Элиаса Дорнелеса (Elias Dorneles), Леонардо Александра Феррейра Лейте (Leonardo Alexandre Ferreira Leite), Брюса Эккеля (Bruce Eckel), Дж. С. Буэно (J. S. Bueno), Рафаэля Гонкальвеса (Rafael Goncalves), Алекса Чиаранда (Alex Chiaranda), Гуто Майа (Guto Maia), Лукаса Видо (Lucas Vido) и Лукаса Бруниальти (Lucas Brunialti).

На протяжении многих лет разные люди побуждали меня написать книгу, но самыми убедительными были Рубенс Пратес (Rubens Prates), Аурелио Жаргас (Aurelio Jargas), Руда Моура (Ruda Moura) и Рубенс Алтимари (Rubens Altimari). Маурицио Буссаб (Mauricio Bussab) открыл мне многие двери и, в частности, поддержал мою первую настоящую попытку написать книгу. Ренцо Нучителли (Renzo Nuccitelli) поддерживал этот проект с самого начала, хотя это и замедлило нашу совместную работу над сайтом *python.pro.br*.

Чудесное сообщество Python в Бразилии состоит из знающих, щедрых и веселых людей. В бразильской группе пользователей Python тысячи членов (*https://groups.google.com/group/python-brasil*), а наши национальные конференции собирают сотни участников, но на меня как на питониста наибольшее влияние оказали Леонардо Рохаель (Leonardo Rochael), Адриано Петрич (Adriano Petrich), Даниэль Вайзенхер (Daniel Vainsencher), Родриго РБП Пиментель (Rodrigo RBP Pimentel), Бруно Гола (Bruno Gola), Леонардо Сантагада (Leonardo Santagada), Джин Ферри (Jean Ferri), Родриго Сенра (Rodrigo Senra), Дж. С. Буэно (J. S. Bueno), Дэвид Кваст (David Kwast), Луис Ирбер (Luiz Irber), Освальдо Сантана (Osvaldo Santana), Фернандо Масанори (Fernando Masanori), Энрике Бастос (Henrique Bastos), Густаво Нимейер (Gustavo Niemayer), Педро Вернек (Pedro Werneck), Густаво Барбиери (Gustavo Barbieri), Лало Мартинс (Lalo Martins), Данило Беллини (Danilo Bellini) и Педро Крогер (Pedro Kroger).

Дорнелес Тремеа был моим большим другом (который щедро делился своим временем и знаниями), замечательным специалистом и самым влиятельным лидером Бразильской ассоциации Python. Он ушел от нас слишком рано.

На протяжении многих лет мои студенты учили меня, задавая вопросы, делясь своими озарениями, мнениями и нестандартными подходами к решению задач. Эрико Андреи (Erico Andrei) и компания Simples Consultoria дали мне возможность посвятить себя преподаванию Python.

Мартин Фаассен (Martijn Faassen) научил меня работать с платформой Grok и поделился бесценными мыслями о Python и неандертальцах. Работа, проделанная им, а также Полом Эвериттом (Paul Everitt), Крисом Макдоно (Chris McDonough), Тресом Сивером (Tres Seaver), Джимом Фултоном (Jim Fulton), Шейном Хэтуэем (Shane Hathaway), Леннартом Негебро (Lennart Regebro), Аланом Руньяном (Alan Runyan), Александром Лими (Alexander Limi), Мартином Питерсом (Martijn Pieters), Годфруа Шапелем (Godefroid Chapelle) и другими участниками проектов Zope, Plone и Pyramid, сыграла решающую роль в моей карьере. Благодаря Zope и серфингу на гребне первой волны веба я в 1998 году получил возможность зарабатывать на жизнь программированием на Python.

Хосе Октавио Кастро Невес (Jose Octavio Castro Neves) стал моим партнером в первой бразильской компании, занимающейся разработкой на Python.

У меня было так много учителей в более широком сообществе пользователей Python, что перечислить их всех нет никакой возможности, но, помимо вышеупомянутых, я в большом долгу перед Стивом Холденом (Steve Holden), Раумондом Хеттингером (Raymond Hettinger), А. М. Кухлингом (A. M. Kuchling), Дэвидом Бизли (David Beazley), Фредриком Лундхом (Fredrik Lundh), Дугом Хеллманом (Doug Hellmann), Ником Кофлином (Nick Coghlan), Марком Пилгримом (Mark Pilgrim), Мартином Питерсом (Martijn Pieters), Брюсом Эккелем (Bruce Eckel), Мишелем Симионато (Michele Simionato), Уэсли Чаном (Wesley Chun), Брэндоном Крейгом Родсом (Brandon Craig Rhodes), Филипом Гуо (Philip Guo), Дэниэлем Гринфилдом (Daniel Greenfeld), Одри Роем (Audrey Roy) и Брэттом Слаткиным (Brett Slatkin), которые подсказали мне, как лучше преподавать Python.

Большая часть книги была написана у меня дома и еще в двух местах: CoffeeLab и Garoa Hacker Clube. CoffeeLab (*http://coffeelab.com.br/*) – квартал любителей кофе в районе Вила Мадалена в бразильском городе Сан-Паулу. Garoa Hacker Clube (*https://garoa.net.br/*) – открытая для всех лаборатория, в которой каждый может бесплатно опробовать новые идеи.

Сообщество Garoa стало для меня источником вдохновения, предоставило инфраструктуру и возможность расслабиться. Надеюсь, Алефу понравится эта книга.

Моя мать, Мария Лучия, и мой отец, Хайро, всегда и во всем поддерживали меня. Я хотел бы, чтобы он мог увидеть эту книгу, я счастлив, что она порадуется ей вместе со мной.

Моя жена, Марта Мелло, 15 месяцев терпела мужа, который был постоянно занят, но не лишала меня своей поддержки и утешения в самые критические моменты, когда мне казалось, что я не выдержу этого марафона.

Спасибо вам всем. За всё.

Часть I

Структуры данных

Глава 1

Модель данных в языке Python

> У Гвидо поразительное эстетическое чувство дизайна языка. Я встречал многих замечательных проектировщиков языков программирования, создававших теоретически красивые языки, которыми никто никогда не пользовался, а Гвидо – один из тех редких людей, которые могут создать язык, немного не дотягивающий до теоретической красоты, зато такой, что писать на нем программы в радость.
>
> – Джим Хагьюнин, автор Jython, соавтор AspectJ, архитектор .Net DLR[1]

Одно из лучших качеств Python – его согласованность. Немного поработав с этим языком, вы уже сможете строить обоснованные и правильные предположения о еще незнакомых средствах.

Однако тем, кто раньше учил другой объектно-ориентированный язык, может показаться странным синтаксис `len(collection)` вместо `collection.len()`. Это кажущаяся несообразность – лишь верхушка айсберга, и если ее правильно понять, то она станет ключом к тому, что мы называем «питонизмами». А сам айсберг называется моделью данных в Python и описывает API, следуя которому, можно согласовать свои объекты с самыми идиоматичными средствами языка.

Можно считать, что модель данных описывает Python как каркас. Она формализует различные структурные блоки языка, в частности последовательности, итераторы, функции, классы, контекстные менеджеры и т. д.

При программировании в любом каркасе мы тратим большую часть времени на реализацию вызываемых каркасом методов. Это справедливо и при использовании модели данных Python для построения новых классов. Интерпретатор Python вызывает специальные методы для выполнения базовых операций над объектами, часто такие вызовы происходят, когда встречается некая синтаксическая конструкция. Имена специальных методов начинаются и заканчиваются двумя знаками подчеркивания. Так, за синтаксической конструкцией `obj[key]` стоит специальный метод `__getitem__`. Для вычисления выражения `my_collection[key]` интерпретатор вызывает метод `my_collection.__getitem__(key)`.

Мы реализуем специальные методы, когда хотим, чтобы наши объекты могли поддерживать и взаимодействовать с базовыми конструкциями языка, а именно:

---

[1] История Jython (*http://hugunin.net/story_of_jython.html*), изложенная в предисловии к книге Samuele Pedroni and Noel Rappin «Jython Essentials» (O'Reilly).

- коллекции;
- доступ к атрибутам;
- итерирование (включая асинхронное итерирование с помощью `async for`);
- перегрузку операторов;
- вызов функций и методов;
- представление и форматирование строк;
- асинхронное программирование с использованием `await`;
- создание и уничтожение объектов;
- управляемые контексты (т. е. блоки `with` и `async with`).

Магические и dunder-методы

На жаргоне специальные методы называют *магическими*, но как мы в разговоре произносим имя конкретного метода, например `__getitem__`? Выражение «dunder-getitem» я услышал от автора и преподавателя Стива Холдена. «Dunder» – это сокращенная форма «двойной подчерк до и после». Поэтому специальные методы называются также *dunder-методами*. В главе «Лексический анализ» *Справочного руководства по Python* имеется предупреждение: «*Любое* использование имен вида `__*__` в любом контексте, отличающееся от явно документированного, может привести к ошибке без какого-либо предупреждения».

Что нового в этой главе

В этой главе немного отличий от первого издания, потому что она представляет собой введение в модель данных в Python, которая давно стабилизировалась. Перечислим наиболее существенные изменения:

- в таблицы в разделе «Сводка специальных методов» добавлены методы, поддерживающие асинхронное программирование и другие новые средства;
- на рис. 1.2 показано использование специальных методов API коллекций, включая абстрактный базовый класс `collections.abc.Collection`, появившийся в версии Python 3.6.

Кроме того, здесь и далее я использую синтаксис *f-строк*, введенный в Python 3.6, который проще читать и зачастую удобнее, чем прежние способы форматирования: метод `str.format()` и оператор `%`.

Использовать нотацию `my_fmt.format()` все еще необходимо, когда `my_fmt` определено не там, где выполняется операция форматирования. Например, если `my_fmt` состоит из нескольких строчек и определено в виде константы или читается из конфигурационного файла либо из базы данных. Тут по-другому не сделаешь, но такие ситуации встречаются нечасто.

Колода карт на Python

Следующий пример очень прост, однако демонстрирует выгоды от реализации двух специальных методов: `__getitem__` и `__len__`.

Пример 1.1. Колода как последовательность карт

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

Прежде всего отметим использование `collections.namedtuple` для конструирования простого класса, представляющего одну карту. Мы используем класс `namedtuple` для построения классов, содержащих только атрибуты и никаких методов, как, например, запись базы данных. В данном примере мы воспользовались им для создания простого представления игральной карты, что продемонстрировано в следующем сеансе оболочки:

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

Но изюминка примера – класс `FrenchDeck`. Совсем короткий, он таит в себе немало интересного. Во-первых, как и для любой стандартной коллекции в Python, для колоды можно вызвать функцию `len()`, которая вернет количество карт в ней:

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

Получить карту из колоды, например первую или последнюю, просто благодаря методу `__getitem__`:

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

Нужно ли создавать метод для выбора случайной карты? Необязательно. В Python уже есть функция выборки случайного элемента последовательности: `random.choice`. Достаточно вызвать ее для экземпляра колоды:

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
```

```
>>> choice(deck)
Card(rank='2', suit='clubs')
```

Мы только что видели два преимущества использования специальных методов для работы с моделью данных.

- Пользователям нашего класса нет нужды запоминать нестандартные имена методов для выполнения стандартных операций («Как мне получить количество элементов? То ли `.size()`, то ли `.length()`, то ли еще как-то»).
- Проще воспользоваться богатством стандартной библиотеки Python (например, функцией `random.choice`), чем изобретать велосипед.

Но это еще не все.

Поскольку метод `__getitem__` делегирует выполнение оператору `[]` объекта `self._cards`, колода автоматически поддерживает срезы. Вот как можно посмотреть три верхние карты в неперетасованной колоде, а затем выбрать только тузы, начав с элемента, имеющего индекс 12, и пропуская по 13 карт:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Стоило нам реализовать специальный метод `__getitem__`, как колода стала допускать итерирование:

```
>>> for card in deck: # doctest: +ELLIPSIS
...   print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

Итерировать можно и в обратном порядке:

```
>>> for card in reversed(deck): # doctest: +ELLIPSIS
...   print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```

Многоточие в тестах

Всюду, где возможно, листинги сеансов оболочки извлекались из doctest-скриптов, чтобы гарантировать точность. Если вывод слишком длинный, то опущенная часть помечается многоточием, как в последней строке показанного выше кода. В таких случаях мы используем директиву `# doctest: +ELLIPSIS`, чтобы тест завершился успешно. Если вы будете вводить эти примеры в интерактивной оболочке, можете вообще опускать директивы doctest.

Итерирование часто подразумевается неявно. Если в коллекции отсутствует метод `__contains__`, то оператор `in` производит последовательный просмотр.

Конкретный пример – в классе `FrenchDeck` оператор `in` работает, потому что этот класс итерируемый. Проверим:

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

А как насчет сортировки? Обычно карты ранжируются по достоинству (тузы – самые старшие), а затем по масти в порядке пики (старшая масть), черви, бубны и трефы (младшая масть). Приведенная ниже функция ранжирует карты, следуя этому правилу: `0` означает двойку треф, а `51` – туз пик.

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

С помощью функции `spades_high` мы теперь можем расположить колоду в порядке возрастания:

```
>>> for card in sorted(deck, key=spades_high): # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 карт опущено)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Хотя класс `FrenchDeck` неявно наследует `object`, его функциональность не наследуется, а является следствием использования модели данных и композиции. Вследствие реализации специальных методов `__len__` и `__getitem__` класс `FrenchDeck` ведет себя как стандартная последовательность и позволяет использовать базовые средства языка (например, итерирование и получение среза), а также функции `reversed` и `sorted`. Благодаря композиции реализации методов `__len__` и `__getitem__` могут перепоручать работу объекту `self._cards` класса `list`.

А как насчет тасования?

В текущей реализации объект класса `FrenchDeck` нельзя перетасовать, потому что он неизменяемый: ни карты, ни их позиции невозможно изменить, не нарушая инкапсуляцию (т. е. манипулируя атрибутом `_cards` непосредственно). В главе 13 мы исправим это, добавив однострочный метод `__setitem__`.

Как используются специальные методы

Говоря о специальных методах, нужно все время помнить, что они предназначены для вызова интерпретатором, а не вами. Вы пишете не `my_object.__len__()`, а `len(my_object)`, и если `my_object` – экземпляр определенного пользователем класса, то Python вызовет реализованный вами метод экземпляра `__len__`.

Однако для встроенных классов, например `list`, `str`, `bytearray`, или расширений типа массивов NumPy интерпретатор поступает проще. Коллекции переменного размера, написанные на C, включают структуру[1] `PyVarObject`, в которой имеется поле `ob_size`, содержащее число элементов в коллекции. Поэтому если `my_object` – экземпляр одного из таких встроенных типов, то `len(my_object)` возвращает значение поля `ob_size`, что гораздо быстрее, чем вызов метода.

Как правило, специальный метод вызывается неявно. Например, предложение `for i in x:` подразумевает вызов функции `iter(x)`, которая, в свою очередь, может вызывать метод `x.__iter__()`, если он реализован, или использовать `x.__getitem__()`, как в примере класса `FrenchDeck`.

Обычно в вашей программе не должно быть много прямых обращений к специальным методам. Если вы не пользуетесь метапрограммированием, то чаще будете реализовывать специальные методы, чем явно вызывать их. Единственный специальный метод, который регулярно вызывается из пользовательского кода напрямую, – `__init__`, он служит для инициализации суперкласса из вашей реализации `__init__`.

Если необходимо обратиться к специальному методу, то обычно лучше вызвать соответствующую встроенную функцию (например, `len`, `iter`, `str` и т. д.). Она вызывает нужный специальный метод и нередко предоставляет дополнительный сервис. К тому же для встроенных типов это быстрее, чем вызов метода. См. раздел «Использование iter совместно с вызываемым объектом» главы 17.

В следующих разделах мы рассмотрим некоторые из наиболее важных применений специальных методов:

- эмуляция числовых типов;
- строковое представление объектов;
- булево значение объекта;
- реализация коллекций.

Эмуляция числовых типов

Несколько специальных методов позволяют объектам иметь операторы, например `+`. Подробно мы рассмотрим этот вопрос в главе 16, а пока проиллюстрируем использование таких методов на еще одном простом примере.

Мы реализуем класс для представления двумерных векторов, обычных евклидовых векторов, применяемых в математике и физике (рис. 1.1).

Для представления двумерных векторов можно использовать встроенный класс `complex`, но наш класс допускает обобщение на *n*-мерные векторы. Мы займемся этим в главе 17.

[1] В языке C структура – это тип записи с именованными полями.

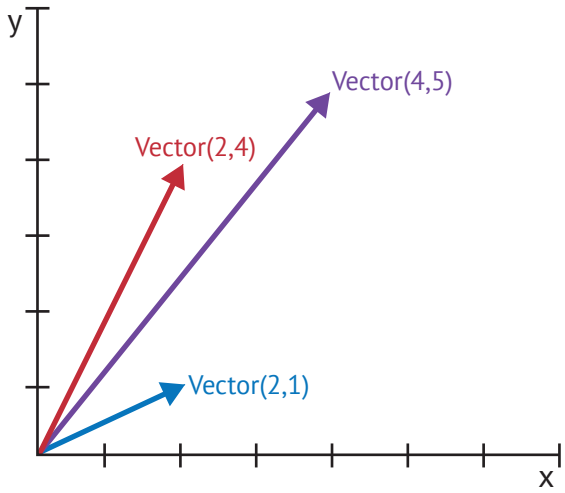

Рис. 1.1. Пример сложения двумерных векторов: Vector(2, 4) + Vector(2, 1) = Vector(4, 5)

Для начала спроектируем API класса, написав имитацию сеанса оболочки, которая впоследствии станет тестом. В следующем фрагменте тестируется сложение векторов, изображенное на рис. 1.1.

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

Отметим, что оператор `+` порождает результат типа `Vector`, который отображается в оболочке интуитивно понятным образом.

Встроенная функция `abs` возвращает абсолютную величину вещественного числа – целого или с плавающей точкой – и модуль числа типа `complex`, поэтому для единообразия наш API также использует функцию `abs` для вычисления модуля вектора:

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

Мы можем еще реализовать оператор `*`, выполняющий умножение на скаляр (т. е. умножение вектора на число, в результате которого получается новый вектор с тем же направлением и умноженным на данное число модулем):

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

В примере 1.2 приведен класс `Vector`, реализующий описанные операции с помощью специальных методов `__repr__`, `__abs__`, `__add__` и `__mul__`.

Пример 1.2. Простой класс двумерного вектора

```
«»»
vector2d.py: упрощенный класс, демонстрирующий некоторые специальные методы.

Упрощен из дидактических соображений. Классу не хватает правильной
обработки ошибок, особенно в методах ``__add__`` and ``__mul__``.

Далее в книге этот пример будет существенно расширен.

Сложение::

    >>> v1 = Vector(2, 4)
    >>> v2 = Vector(2, 1)
    >>> v1 + v2
    Vector(4, 5)

Абсолютная величина::

    >>> v = Vector(3, 4)
    >>> abs(v)
    5.0

Умножение на скаляр::

    >>> v * 3
    Vector(9, 12)
    >>> abs(v * 3)
    15.0

"""

import math
class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x!r}, {self.y!r})'

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

Мы реализовали пять специальных методов, помимо хорошо знакомого `__init__`. Отметим, что ни один из них не вызывается напрямую внутри само-

го класса или при типичном использовании класса, показанном в листингах сеансов оболочки. Как уже было сказано, чаще всего специальные методы вызывает интерпретатор Python.

В примере 1.2 реализовано два оператора: `+` и `*`, чтобы продемонстрировать использование методов `__add__` и `__mul__`. В обоих случаях метод создает и возвращает новый экземпляр класса `Vector`, не модифицируя ни один операнд – аргументы `self` и `other` только читаются. Именно такого поведения ожидают от инфиксных операторов: создавать новые объекты, не трогая операндов. Мы еще вернемся к этому вопросу в главе 16.

Реализация в примере 1.2 позволяет умножать `Vector` на число, но не число на `Vector`. Это нарушает свойство коммутативности операции умножения на скаляр. В главе 16 мы исправим данный недостаток, реализовав специальный метод `__rmul__`.

В следующих разделах мы обсудим другие специальные методы класса `Vector`.

Строковое представление

Специальный метод `__repr__` вызывается встроенной функцией `repr` для получения строкового представления объекта. Если бы мы не реализовали метод `__repr__`, то объект класса `Vector` был бы представлен в оболочке строкой вида `<Vector object at 0x10e100070>`.

Интерактивная оболочка и отладчик вызывают функцию `repr`, передавая ей результат вычисления выражения. То же самое происходит при обработке спецификатора `%r` в случае классического форматирования с помощью оператора `%` и при обработке поля преобразования `!r` в новом синтаксисе форматной строки (*https://docs.python.org/3.10/library/string.html#format-string-syntax*), применяемом в *f-строках* в методе `str.format`.

Отметим, что в *f-строке* в нашей реализации метода `__repr__` мы использовали `!r` для получения стандартного представления отображаемых атрибутов. Это разумный подход, потому что в нем отчетливо проявляется существенное различие между `Vector(1, 2)` и `Vector('1', '2')` – второй вариант в контексте этого примера не заработал бы, потому что аргументами конструктора должны быть числа, а не объекты `str`.

Строка, возвращаемая методом `__repr__`, должна быть однозначно определена и по возможности соответствовать коду, необходимому для восстановления объекта. Именно поэтому мы выбрали представление, напоминающее вызов конструктора класса (например, `Vector(3, 4)`).

В отличие от `__repr__`, метод `__str__` вызывается конструктором `str()` и неявно используется в функции `print`. Он должен возвращать строку, пригодную для показа пользователям.

Иногда строка, возвращенная методом `__repr__`, уже пригодна для показа пользователям, тогда нет нужды писать метод `__str__`, т. к. реализация, унаследованная от класса `object`, вызывает `__repr__`, если нет альтернативы. Пример 5.2 – один из немногих в этой книге, где используется пользовательский метод `__str__`.

Программисты, имеющие опыт работы с языками, где имеется метод `toString`, по привычке реализуют метод `__str__`, а не `__repr__`. Если вы реализуете только один из этих двух методов, то пусть это будет `__repr__`.

На сайте Stack Overflow был задан вопрос «What is the difference between __str__ and __repr__ in Python» (*https://stackoverflow.com/questions/1436703/what-is-the-difference-between-str-and-repr*), ответ на который содержит прекрасные разъяснения Алекса Мартелли и Мартина Питерса.

Булево значение пользовательского типа

Хотя в Python есть тип `bool`, интерпретатор принимает любой объект в булевом контексте, например в условии `if`, в управляющем выражении цикла `while` или в качестве операнда операторов `and`, `or` и `not`. Чтобы определить, является ли выражение истинным или ложным, применяется функция `bool(x)`, которая возвращает `True` или `False`.

По умолчанию любой экземпляр пользовательского класса считается истинным, но положение меняется, если реализован хотя бы один из методов `__bool__` или `__len__`. Функция `bool(x)`, по существу, вызывает `x.__bool__()` и использует полученный результат. Если метод `__bool__` не реализован, то Python пытается вызвать `x.__len__()` и при получении нуля функция `bool` возвращает `False`. В противном случае `bool` возвращает `True`.

Наша реализация `__bool__` концептуально проста: метод возвращает `False`, если модуль вектора равен 0, и `True` в противном случае. Для преобразования модуля в булеву величину мы вызываем `bool(abs(self))`, поскольку ожидается, что метод `__bool__` возвращает булево значение. Вне метода `__bool__` редко возникает надобность вызывать `bool()` явно, потому что любой объект можно использовать в булевом контексте.

Обратите внимание на то, как специальный метод `__bool__` обеспечивает согласованность пользовательских объектов с правилами проверки значения истинности, определенными в главе «Встроенные типы» документации по стандартной библиотеке Python (*http://docs.python.org/3/library/stdtypes.html#truth*).

Можно было бы написать более быструю реализацию метода `Vector.__bool__`:

```
    def __bool__(self):
        return bool(self.x or self.y)
```

Она сложнее воспринимается, зато позволяет избежать обращений к `abs` и `__abs__`, возведения в квадрат и извлечения корня. Явное преобразование в тип `bool` необходимо, потому что метод `__bool__` должен возвращать булево значение, а оператор `or` возвращает один из двух операндов: результат вычисления `x or y` равен `x`, если `x` истинно, иначе равен `y` вне зависимости от его значения.

API коллекций

На рис. 1.2 описаны интерфейсы основных типов коллекций, поддерживаемых языком. Все классы на этой диаграмме являются *абстрактными базо-*

выми классами, ABC. Такие классы и модуль `collections.abc` рассматриваются в главе 13. Цель этого краткого раздела – дать общее представление о самых важных интерфейсах коллекций в Python и показать, как они строятся с помощью специальных методов.

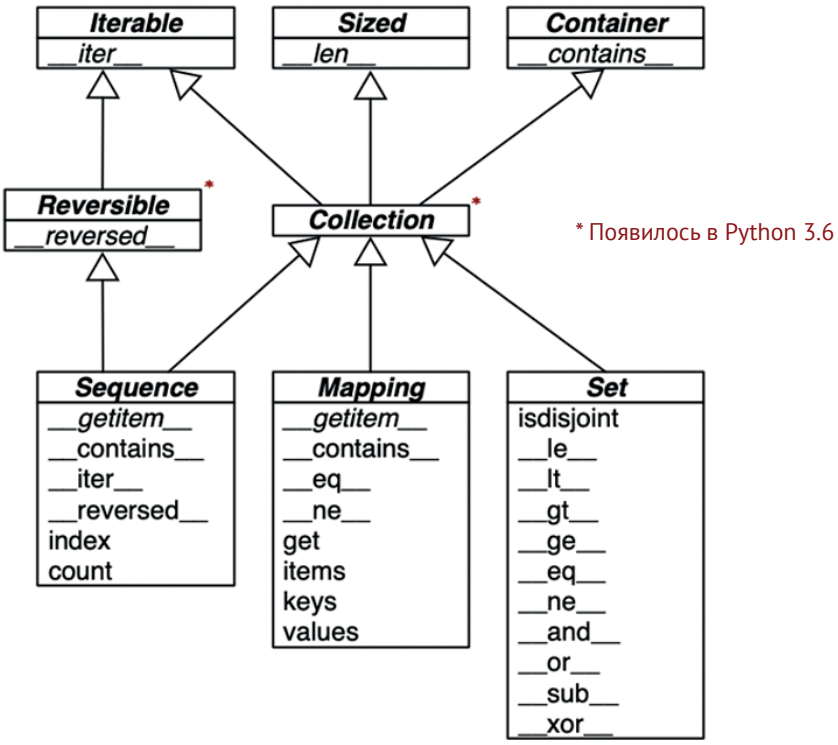

Рис. 1.2. UML-диаграмма классов, иллюстрирующая наиболее важные типы коллекций. Методы, имена которых набраны курсивом, абстрактные, поэтому должны быть реализованы в конкретных подклассах, например `list` и `dict`. У остальных методов имеются конкретные реализации, которые подклассы могут унаследовать

У каждого из ABC в верхнем ряду есть всего один специальный метод. Абстрактный базовый класс `Collection` (появился в версии Python 3.6) унифицирует все три основных интерфейса, который должна реализовать любая коллекция:

- `Iterable` для поддержки `for`, распаковки и других видов итерирования;
- `Sized` для поддержки встроенной функции `len`;
- `Container` для поддержки оператора `in`.

Python не требует, чтобы конкретные классы наследовали какому-то из этих ABC. Любой класс, реализующий метод `__len__`, удовлетворяет требованиям интерфейса `Sized`.

Перечислим три важнейшие специализации `Collection`:

- `Sequence`, формализует интерфейс встроенных классов, в частности `list` и `str`;
- `Mapping`, реализован классами `dict`, `collections.defaultdict` и др.;
- `Set`, интерфейс встроенных типов `set` и `frozenset`.

Только `Sequence` реализует интерфейс `Reversible`, потому что последовательности поддерживают произвольное упорядочение элементов, тогда как отображения и множества таким свойством не обладают.

Начиная с версии Python 3.7 тип `dict` официально считается «упорядоченным», но это лишь означает, что порядок вставки ключей сохраняется. Переупорядочить ключи словаря `dict` так, как вам хочется, невозможно.

Все специальные методы ABC `Set` предназначены для реализации инфиксных операторов. Например, выражение `a & b` вычисляет пересечение множеств `a` и `b` и реализовано специальным методом `__and__`.

В следующих двух главах мы подробно рассмотрим стандартные библиотечные последовательности, отображения и множества.

А пока перейдем к основным категориям специальных методов, определенным в модели данных Python.

Сводка специальных методов

В главе «Модель данных» (*http://docs.python.org/3/reference/datamodel.html*) справочного руководства по языку Python перечислено более 80 специальных методов. Больше половины из них используются для реализации операторов: арифметических, поразрядных и сравнения. Следующие таблицы дают представление о том, что имеется в нашем распоряжении.

В табл. 1.1 показаны имена специальных методов, за исключением тех, что используются для реализации инфиксных операторов и базовых математических функций, например `abs`. Большинство этих методов будут рассмотрены на протяжении книги, в т. ч. недавние добавления: асинхронные специальные методы, в частности `__anext__` (добавлен в Python 3.5), и точка подключения для настройки класса `__init_subclass__` (добавлен в Python 3.6).

Таблица 1.1. Имена специальных методов (операторы не включены)

Категория	Имена методов
Представление в виде строк и байтов	`__repr__`, `__str__`, `__format__`, `__bytes__`, `__fspath__`
Преобразование в число	`__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`, `__index__`
Эмуляция коллекций	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
Итерирование	`__iter__`, `__aiter__`, `__next__`, `__anext__`, `__reversed__`
Выполнение объектов, допускающих вызов, или сопрограмм	`__call__`, `__await__`
Управление контекстом	`__enter__`, `__exit__`, `__aenter__`, `__aexit__`
Создание и уничтожение объектов	`__new__`, `__init__`, `__del__`

Окончание табл. 1.1

Категория	Имена методов
Управление атрибутами	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`
Дескрипторы атрибутов	`__get__`, `__set__`, `__delete__`, `__set_name__`
Абстрактные базовые классы	`__instancecheck__`, `__subclasscheck__`
Метапрограммирование классов	`__prepare__`, `_init_subclass__`, `__class_getitem__`, `__mro_entries__`

Инфиксные и числовые операторы поддерживаются специальными методами, перечисленными в табл. 1.2. В версии Python 3.5 были добавлены методы `__matmul__`, `__rmatmul__` и `__imatmul__` для поддержки `@` в роли инфиксного оператора умножения матриц (см. главу 16).

Таблица 1.2. Имена специальных методов для операторов

Категория операторов	Символы	Имена методов
Унарные числовые операторы	`- + abs()`	`__neg__ __pos__ __abs__`
Операторы сравнения	`< <= == !- > >=`	`__lt__ __le__ __eq__ __ne__ __gt__ __ge__`
Арифметические операторы	`+ - * / // % @ divmod() round() ** pow()`	`__add__ __sub__ __mul__ __truediv__ __floordiv__ __mod__ __matmul__ __divmod__ __round__ __pow__`
Инверсные арифметические операторы	(арифметические операторы с переставленными операндами)	`__radd__ __rsub__ __rmul__ __rtruediv__ __rfloordiv__ __rmod__ __rmatmul__ __rdivmod__ __rpow__`
Арифметические операторы составного присваивания	`+= -= *= /= //= %= @= **=`	`__iadd__ __isub__ __imul__ __itruediv____ifloordiv__ __imod__ __imatmul__ __ipow__`
Поразрядные операторы	`& | ^ << >> ~`	`__and__ __or__ __xor__ __lshift__ __rshift__ __invert__`
Инверсные поразрядные операторы	(поразрядные операторы с переставленными операндами)	`__rand__ __ror__ __rxor__ __rlshift__ __rrshift__`
Поразрядные операторы составного присваивания	`&= |= ^= <<= >>=`	`__iand__ __ior__ __ixor__ __ilshift__ __irshift__`

Python вызывает инверсный специальный метод от имени второго операнда, если нельзя использовать соответственный специальный метод от имени первого операнда. Операторы составного присваивания – сокращенный способ вызвать инфиксный оператор с последующим присваиванием переменной, например `a += b`.

В главе 16 инверсные операторы и составное присваивание рассматриваются подробнее.

Почему len – не метод

Я задавал этот вопрос разработчику ядра Раймонду Хэттингеру в 2013 году, смысл его ответа содержится в цитате из «Дзен Python»: «практичность важнее чистоты» (*https://www.python.org/doc/humor/#thezen-of-python*). В разделе «Как используются специальные методы» выше я писал, что функция `len(x)` работает очень быстро, если `x` – объект встроенного типа. Для встроенных объектов интерпретатор CPython вообще не вызывает никаких методов: длина просто читается из поля C-структуры. Получение количества элементов в коллекции – распространенная операция, которая должна работать эффективно для таких разных типов, как `str`, `list`, `memoryview` и т. п.

Иначе говоря, `len` не вызывается как метод, потому что играет особую роль в модели данных Python, равно как и `abs`. Но благодаря специальному методу `__len__` можно заставить функцию `len` работать и для пользовательских объектов. Это разумный компромисс между желанием обеспечить как эффективность встроенных объектов, так и согласованность языка. Вот еще цитата из «Дзен Python»: «особые случаи не настолько особые, чтобы из-за них нарушать правила».

Если рассматривать `abs` и `len` как унарные операторы, то, возможно, вы простите их сходство с функциями, а не с вызовами метода, чего следовало бы ожидать от ОО-языка. На самом деле в языке ABC – непосредственном предшественнике Python, в котором впервые были реализованы многие его средства, – существовал оператор `#`, эквивалентный `len` (следовало писать `#s`). При использовании в качестве инфиксного оператора – `x#s` – он подсчитывал количество вхождений `x` и `s`; в Python для этого нужно вызвать `s.count(x)`, где `s` – произвольная последовательность.

Резюме

Благодаря реализации специальных методов пользовательские объекты могут вести себя как встроенные типы. Это позволяет добиться выразительного стиля кодирования, который сообщество считает «питоническим».

Важное требование к объекту Python – обеспечить полезные строковые представления себя: одно – для отладки и протоколирования, другое – для показа пользователям. Именно для этой цели предназначены специальные методы `__repr__` и `__str__`.

Эмуляция последовательностей, продемонстрированная на примере класса `FrenchDeck`, – одно из самых распространенных применений специальных методов. Устройство большинства типов последовательностей – тема главы 2, а реализация собственных последовательностей будет рассмотрена в главе 12 в контексте создания многомерного обобщения класса `Vector`.

Благодаря перегрузке операторов Python предлагает богатый набор числовых типов, от встроенных до `decimal.Decimal` и `fractions.Fraction`, причем все они поддерживают инфиксные арифметические операторы. Библиотека анализа данных NumPy поддерживает инфиксные операторы для матриц и тензоров. Реализация операторов, в том числе инверсных и составного присваивания, будет продемонстрирована в главе 16 в процессе расширения класса `Vector`.

Использование и реализация большинства других специальных методов, входящих в состав модели данных Python, рассматривается в разных частях книги.

Дополнительная литература

Глава «Модель данных» (*http://docs.python.org/3/reference/datamodel.html*) справочного руководства по языку Python – канонический источник информации по теме этой главы и значительной части изложенного в книге материала.

В книге Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», третье издание (O'Reilly), прекрасно объясняется модель данных. Данное ими описание механизма доступа к атрибутам – самое полное из всех, что я видел, если не считать самого исходного кода CPython на C. Мартелли также очень активен на сайте Stack Overflow, ему принадлежат более 6200 ответов. С его профилем можно ознакомиться по адресу *http://stackoverflow.com/users/95810/alex-martelli*.

Дэвид Бизли написал две книги, в которых подробно описывается модель данных в контексте Python 3: «Python Essential Reference», издание 4 (Addison-Wesley Professional), и «*Python Cookbook*»[1], издание 3 (O'Reilly), в соавторстве с Брайаном Л. Джонсом.

В книге Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow «The Art of the Metaobject Protocol» (MIT Press) объясняется протокол метаобъектов, одним из примеров которого является модель данных в Python.

Поговорим

Модель данных или объектная модель?

То, что в документации по Python называется «моделью данных», большинство авторов назвали бы «объектной моделью Python». В книгах Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», издание 3, и David Beazley «Python Essential Reference», издание 4, – лучших книгах по «модели данных Python» – употребляется термин «объектная модель». В Википедии самое первое определение модели данных (*http://en.wikipedia.org/wiki/Object_model*) звучит так: «Общие свойства объектов в конкретном языке программирования». Именно в этом и заключается смысл «модели данных Python». В этой книге я употребляю термин «модель данных», потому что его предпочитают авторы документации и потому что так называется глава в справочном руководстве по языку Python (*https://docs.python.org/3/reference/datamodel.html*), имеющая прямое касательство к нашему обсуждению.

Магические методы

В словаре «The Original Hacker's Dictionary» (*https://www.dourish.com/goodies/jargon.html*) термин *магический* определяется как «еще не объясненный или слишком сложный для объяснения» или как «не раскрываемый публично механизм, позволяющий делать то, что иначе было бы невозможно».

В сообществе Ruby эквиваленты специальных методов называют магическими. Многие пользователи из сообщества Python также восприняли этот термин. Лично я считаю, что специальные методы – прямая противоположность магии. В этом отношении языки Python и Ruby одинаковы: тот и другой предоставляют развитый протокол метаобъектов, отнюдь не магический, но позволяющий пользователям применять те же средства, что доступны разра-

[1] *Бизли Д., Джонс Б. К.* Python. Книга рецептов. М.: ДМК Пресс, 2019 // *https://dmkpress.com/catalog/computer/programming/python/978-5-97060-751-0/*

ботчикам ядра, которые пишут интерпретаторы этих языков.

Сравним это с Go. В этом языке у некоторых объектов есть действительно магические возможности, т. е. такие, которые невозможно имитировать в пользовательских типах. Например, массивы, строки и отображения в Go поддерживают использование квадратных скобок для доступа к элементам: `a[i]`. Но не существует способа приспособить нотацию `[]` к новым, определенным пользователем типам. Хуже того, в Go нет ни понятия интерфейса итерируемости, ни объекта итератора на пользовательском уровне, поэтому синтаксическая конструкция `for/range` ограничена поддержкой пяти «магических» встроенных типов, в т. ч. массивов, строк и отображений.

Быть может, в будущем проектировщики Go расширят протокол метаобъектов. А пока в нем куда больше ограничений, чем в Python и Ruby.

Метаобъекты

«The Art of the Metaobject Protocol» (AMOP) – моя любимая книга по компьютерам. Но и отбросив в сторону субъективизм, термин «протокол метаобъектов» полезен для размышления о модели данных в Python и о похожих средствах в других языках. Слово «метаобъект» относится к объектам, являющимся структурными элементами самого языка. А «протокол» в этом контексте – синоним слова «интерфейс». Таким образом, протокол метаобъектов – это причудливый синоним «объектной модели»: API для доступа к базовым конструкциям языка.

Развитый протокол метаобъектов позволяет расширять язык для поддержки новых парадигм программирования. Грегор Кикзалес, первый автор книги AMOP, впоследствии стал первопроходцем аспектно-ориентированного программирования и первоначальным автором AspectJ, расширения Java для реализации этой парадигмы. В динамическом языке типа Python реализовать аспектно-ориентированное программирование гораздо проще, и существует несколько каркасов, в которых это сделано. Самым известным из них является каркас *zope.interface* (*http://docs.zope.org/zope.interface/*), на базе которого построена система управления контентом Plone (*https://plone.org/*).

Глава 2

Массив последовательностей

> Как вы, наверное, заметили, некоторые из упомянутых операций одинаково работают для текстов, списков и таблиц. Для текстов, списков и таблиц имеется обобщенное название «ряд» [...]. Команда FOR также единообразно применяется ко всем рядам.
>
> – Leo Geurts, Lambert Meertens, Steven Pembertonm, «ABC Programmer's Handbook»[1]

До создания Python Гвидо принимал участие в разработке языка ABC. Это был растянувшийся на 10 лет исследовательский проект по проектированию среды программирования для начинающих. В ABC первоначально появились многие идеи, которые мы теперь считаем «питоническими»: обобщенные операции с последовательностями, встроенные типы кортежа и отображения, структурирование кода с помощью отступов, строгая типизация без объявления переменных и др. Не случайно Python так дружелюбен к пользователю.

Python унаследовал от ABC единообразную обработку последовательностей. Строки, списки, последовательности байтов, массивы, элементы XML, результаты выборки из базы данных – все они имеют общий набор операций, включающий итерирование, получение среза, сортировку и конкатенацию.

Зная о различных последовательностях, имеющихся в Python, вы не станете изобретать велосипед, а наличие общего интерфейса побуждает создавать API, которые согласованы с существующими и будущими типами последовательностей.

Материал этой главы в основном относится к последовательностям вообще: от знакомых списков `list` до типов `str` и `bytes`, появившихся в Python 3. Здесь же будет рассмотрена специфика списков, кортежей, массивов и очередей, однако обсуждение строк Unicode и последовательностей байтов мы отложим до главы 4. Кроме того, здесь мы рассматриваем только готовые типы последовательностей, а о том, как создавать свои собственные, поговорим в главе 12.

В этой главе будут рассмотрены следующие темы:

- списковые включения и основы генераторных выражений;
- использование кортежей как записей и как неизменяемых списков;
- распаковка последовательностей и последовательности-образцы;
- чтение и запись срезов;
- специализированные типы последовательностей, в частности массивы и очереди.

---

[1] Leo Geurts, Lambert Meertens, Steven Pemberton «ABC Programmer's Handbook», стр. 8 (Bosko Books).

Что нового в этой главе

Самое важное новшество в этой главе – раздел «Сопоставление с последовательностями-образцами». Это первое знакомство с новым механизмом сопоставления с образцами, появившимся в Python 3.10, на страницах данной книги.

Другие изменения – не столько новшества, сколько улучшения первого издания:

- новая диаграмма и описание внутреннего механизма последовательностей, в котором противопоставляются контейнеры и плоские последовательности;
- краткое сравнение характеристик производительности и потребления памяти классов `list` и `tuple`;
- подводные камни, связанные с изменяемыми элементами, и как их обнаружить в случае необходимости.

Я перенес рассмотрение именованных кортежей в раздел «Классические именованные кортежи» главы 5, где они сравниваются с `typing.NamedTuple` и `@dataclass`.

Чтобы расчистить место для нового материала и не увеличивать объем книги сверх разумного, раздел «Средства работы с упорядоченными последовательностями в модуле bisect», присутствовавший в первом издании, теперь оформлен в виде статьи на сопроводительном сайте *fluentpython.com*.

Общие сведения о встроенных последовательностях

Стандартная библиотека предлагает богатый выбор типов последовательностей, реализованных на C:

Контейнерные последовательности

Позволяют хранить элементы разных типов, в т. ч. вложенные контейнеры. Примерами могут служить `list`, `tuple` и `collections.deque`.

Плоские последовательности

Позволяют хранить элементы только одного типа. Примерами могут служить `str`, `bytes` и `array.array`.

В *контейнерных последовательностях* хранятся ссылки на объекты любого типа, тогда как в *плоских последовательностях* – сами значения прямо в памяти, занятой последовательностью, а не как отдельные объекты Python. См. рис. 2.1.

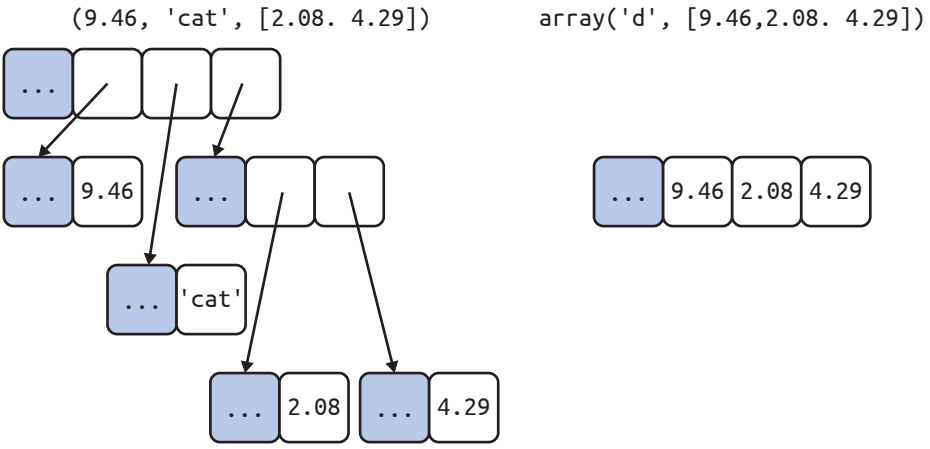

Рис. 2.1. Упрощенные диаграммы размещения кортежа и массива в памяти (тот и другой содержат три элемента). Закрашенные ячейки представляют заголовок объекта Python в памяти – пропорции не соблюдены. В кортеже хранится массив ссылок на элементы. Каждый элемент – отдельный объект Python, быть может, содержащий ссылки на другие объекты, скажем список из двух элементов. Напротив, массив в Python – это единственный объект, в котором хранится С-массив трех чисел типа double

Поэтому плоские последовательности компактнее, но могут содержать только значения примитивных машинных типов, например байты, целые числа и числа с плавающей точкой.

Каждый объект Python, находящийся в памяти, имеет заголовок с метаданными. У простейшего объекта, `float`, имеется поле значения и два поля метаданных:

- `ob_refcnt`: счетчик ссылок на объект;
- `ob_type`: указатель на тип объекта;
- `ob_fval`: число типа `double` (в смысле С), в котором хранится значение с плавающей точкой.

В 64-разрядной сборке Python каждое из этих полей занимает 8 байт. Потому-то массив `float` гораздо компактнее, чем кортеж `float`: массив – это один объект, хранящий сами значения с плавающей точкой, а кортеж состоит из нескольких объектов: сам кортеж и все содержащиеся в нем объекты типа `float`.

Последовательности можно также классифицировать по признаку изменяемости:

Изменяемые последовательности

Например, `list`, `bytearray`, `array.array` и `collections.deque`.

Неизменяемые последовательности

Например, `tuple`, `str` и `bytes`.

На рис. 2.2 показано, что изменяемые последовательности наследуют от неизменяемых все методы и реализуют несколько дополнительных. Встроенные

конкретные типы последовательностей не являются подклассами показанных на рисунке абстрактных базовых классов (ABC) `Sequence` и `MutableSequence`. На самом деле они являются *виртуальными подклассами*, зарегистрированными в этих ABC, как мы увидим в главе 13. Будучи виртуальными подклассами, `tuple` и `list` проходят следующие тесты:

```
>>> from collections import abc
>>> issubclass(tuple, abc.Sequence)
True
>>> issubclass(list, abc.MutableSequence)
True
```

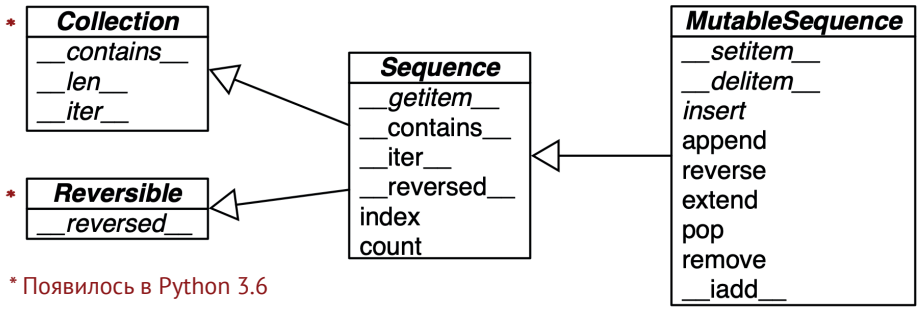

Рис. 2.2. Упрощенная UML-диаграмма нескольких классов из модуля `collections.abc` суперклассы показаны слева, стрелки ведут от подклассов к суперклассам, курсивом набраны имена абстрактных классов и абстрактных методов)

Помнить об этих общих характеристиках – изменяемый и неизменяемый, контейнерная и плоская последовательность – полезно для экстраполяции знаний об одних последовательностях на другие.

Самый фундаментальный тип последовательности – список `list`, изменяемый контейнер. Не сомневаюсь, что вы уверенно владеете списками, поэтому перейдем прямо к списковому включению (list comprehension), эффективному способу построения списков, который недостаточно широко используется из-за незнакомого синтаксиса. Овладение механизмом спискового включения открывает двери к генераторным выражениям, которые – среди прочего – могут порождать элементы для заполнения последовательностей любого типа. То и другое обсуждается в следующем разделе.

Списковое включение и генераторные выражения

Чтобы быстро построить последовательность, можно воспользоваться списковым включением (если конечная последовательность – список) или генераторным выражением (для всех прочих типов последовательностей). Если вы не пользуетесь этими средствами в повседневной работе, клянусь, вы упускаете возможность писать код, который одновременно является и более быстрым, и более удобочитаемым.

Если сомневаетесь насчет «большей удобочитаемости», читайте дальше. Я попробую вас убедить.

Многие программисты для краткости называют списковое включение *listcomp*, а генераторное выражение – *genexp*. Я тоже иногда буду употреблять эти слова.

Списковое включение и удобочитаемость

Вот вам тест: какой код кажется более понятным – в примере 2.1 или 2.2?

Пример 2.1. Построить список кодовых позиций Unicode по строке

```
>>> symbols = '$¢£¥€¤'
>>> codes = []
>>> for symbol in symbols:
... codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Пример 2.2. Построить список кодовых позиций Unicode по строке с применением listcomp

```
>>> symbols = '$¢£¥€¤'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Всякий, кто хоть немного знаком с Python, сможет прочитать пример 2.1. Но после того, как я узнал о списковом включении, пример 2.2 стал казаться мне более удобочитаемым, потому что намерение программиста в нем выражено отчетливее.

Цикл `for` можно использовать с самыми разными целями: просмотр последовательности для подсчета или выборки элементов, вычисление агрегатов (суммы, среднего) и т. д. Так, код в примере 2.1 строит список. А у спискового включения только одна задача – построить новый список, ничего другого оно не умеет.

Разумеется, списковое включение можно использовать и во вред, так что код станет абсолютно непонятным. Я встречал код на Python, в котором listcomp'ы применялись просто для повторения блока кода ради его побочного эффекта. Если вы ничего не собираетесь делать с порожденным списком, то не пользуйтесь этой конструкцией. Кроме того, не переусердствуйте: если списковое включение занимает больше двух строчек, то, быть может, лучше разбить его на части или переписать в виде старого доброго цикла `for`. Действуйте по ситуации: в Python, как и в любом естественном языке, не существует твердых и однозначных правил для написания ясного текста.

Замечание о синтаксисе

В программе на Python переход на другую строку внутри пар скобок [], {} и () игнорируется. Поэтому при построении многострочных списков, списковых включений, генераторных выражений, словарей и прочего можно обходиться без косой черты \ для экранирования символа новой строки, которая к тому же не работает, если после нее случайно поставлен пробел. Кроме того, если эти пары ограничителей используются для определе-

ния литерала с последовательности элементов, перечисленных через запятую, то завершающая запятая игнорируется. Например, при записи многострочного спискового литерала будет мудро поставить после последнего элемента запятую, чтобы потом было проще добавить в конец списка дополнительные элементы и не загромождать лишними строками дельты двух файлов.

Локальная область видимости внутри включений и генераторных выражений

В Python 3.1 у списковых включений, генераторных выражений, а также у родственных им словарных и множественных включений имеется локальная область видимости для хранения переменных, которым присвоено значение в части for.

Однако переменные, которым присвоено значение в операторе `:=`, остаются доступными и после возврата из включения или выражения – в отличие от локальных переменных, определенных в функции. В документе PEP 572 «Assignment Expressions» (*https://peps.python.org/pep-0572/*) область видимости оператора `:=` определена как объемлющая функция, если только соответствующая переменная не является частью объявления `global` или `nonlocal`[1].

```
>>> x = 'ABC'
>>> codes = [ord(x) for x in x]
>>> x ❶
'ABC'
>>> codes
[65, 66, 67]
>>> codes = [last := ord(c) for c in x]
>>> last ❷
67
>>> c ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

❶ Значение `x` не перезаписано: оно по-прежнему привязано к `'ABC'`.

❷ `last` осталось таким, как прежде.

❸ `c` пропала, она существовала только внутри listcomp.

Списковое включение строит список из последовательности или любого другого итерируемого типа путем фильтрации и трансформации элементов. То же самое можно было бы сделать с помощью встроенных функций `filter` и `map`, но, как мы увидим ниже, удобочитаемость при этом пострадает.

Сравнение спискового включения с map и filter

Списковое включение может делать все, что умеют функции `map` и `filter`, без дополнительных выкрутасов, связанных с использованием лямбда-выражений. Взгляните на пример 2.3.

[1] Спасибо читательнице Тине Лапин, указавшей на этот момент.

Пример 2.3. Один и тот же список, построенный с помощью listcomp и композиции `map` и `filter`

```
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord,
symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

Раньше я думал, что композиция `map` и `filter` быстрее эквивалентного спискового включения, но Алекс Мартелли показал, что это не так, по крайней мере в примере выше. В репозитории кода для этой книги имеется скрипт (*https://github.com/fluentpython/example-code-2e/blob/master/02-array-seq/listcomp_speed.py*) *02-array-seq/listcomp_speed.py* для сравнения времени работы listcomp и `filter`/`map`.

В главе 7 я еще вернусь к функциям `map` и `filter`. А пока займемся использованием спискового включения для вычисления декартова произведения: списка, содержащего все кортежи, включающие по одному элементу из каждого списка-сомножителя.

Декартовы произведения

С помощью спискового включения можно сгенерировать список элементов декартова произведения двух и более итерируемых объектов. Декартово произведение – это множество кортежей, включающих по одному элементу из каждого объекта-сомножителя. Длина результирующего списка равна произведению длин входных объектов. См. рис. 2.3.

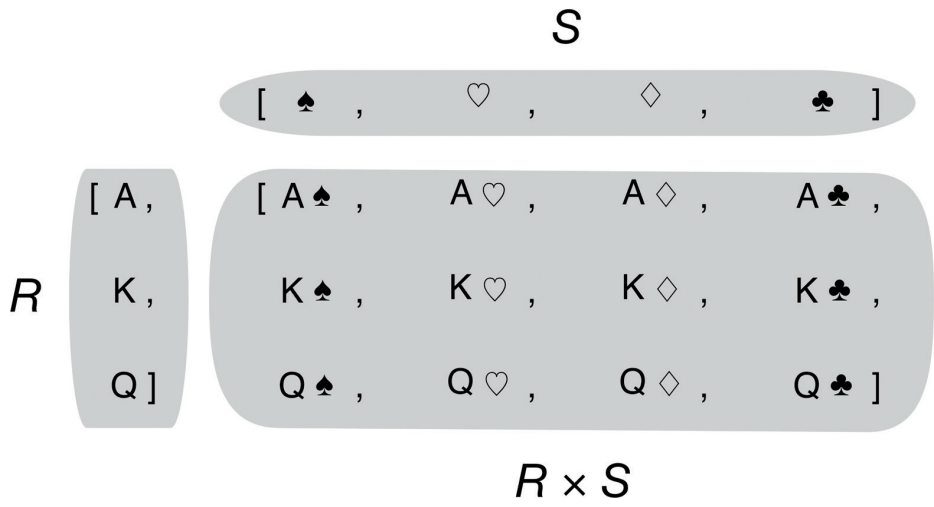

Рис. 2.3. Декартово произведение последовательности трех достоинств карт и последовательности четырех мастей дает последовательность, состоящую из двенадцати пар

Пусть, например, требуется построить список футболок, доступных в двух цветах и трех размерах. В примере 2.4 показано, как это сделать с помощью listcomp. Результирующий список содержит шесть элементов.

Пример 2.4. Построение декартова произведения с помощью спискового включения

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in sizes] ❶
>>> tshirts
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors: ❷
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes  ❸
... for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]
```

❶ Генерирует список кортежей, упорядоченный сначала по цвету, а затем по размеру.

❷ Обратите внимание, что результирующий список упорядочен так, как если бы циклы `for` были вложены именно в том порядке, в котором указаны в списковом включении.

❸ Чтобы расположить элементы сначала по размеру, а затем по цвету, нужно просто поменять местами предложения `for`; после переноса второго предложения `for` на другую строку стало понятнее, как будет упорядочен результат.

В примере 1.1 (глава 1) показанное ниже выражение использовалось для инициализации колоды карт списком, состоящим из 52 карт четырех мастей по 13 карт в каждой:

```
self._cards = [Card(rank, suit) for suit in self.suits
                                 for rank in self.ranks]
```

Списковые включения умеют делать всего одну вещь: строить списки. Для порождения последовательностей других типов придется обратиться к генераторным выражениям. В следующем разделе кратко описывается применение генераторных выражений для построения последовательностей, отличных от списков.

Генераторные выражения

Инициализацию кортежей, массивов и других последовательностей тоже можно начать с использования спискового включения, но genexp экономит память, т. к. отдает элементы по одному, применяя протокол итератора, вместо того чтобы сразу строить целиком список для передачи другому конструктору.

Синтаксически генераторное выражение выглядит так же, как списковое включение, только заключается не в квадратные скобки, а в круглые.

Ниже приведены простые примеры использования генераторных выражений для построения кортежа и массива.

Пример 2.5. Инициализация кортежа и массива с помощью генераторного выражения

```
>>> symbols = '$¢£¥€¤'
>>> tuple(ord(symbol) for symbol in symbols) ❶
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols)) ❷
array('I', [36, 162, 163, 165, 8364, 164])
```

❶ Если генераторное выражение – единственный аргумент функции, то дублировать круглые скобки необязательно.

❷ Конструктор массива принимает два аргумента, поэтому скобки вокруг генераторного выражения обязательны. Первый аргумент конструктора `array` определяет тип хранения чисел в массиве, мы вернемся к этому вопросу в разделе «Массивы» ниже.

В примере 2.6 генераторное выражение используется для порождения декартова произведения и последующей распечатки ассортимента футболок двух цветов и трех размеров. В отличие от примера 2.4, этот список футболок ни в какой момент не находится в памяти: генераторное выражение отдает циклу `for` по одному элементу. Если бы списки, являющиеся сомножителями декартова произведения, содержали по 1000 элементов, то применение генераторного выражения позволило бы сэкономить память за счет отказа от построения списка из миллиона элементов с единственной целью его обхода в цикле `for`.

Пример 2.6. Порождение декартова произведения генераторным выражением

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> for tshirt in (f'{c} {s}' for c in colors for s in sizes): ❶
...     print(tshirt)
...
black S
black M
black L
white S
white M
white L
```

❶ Генераторное выражение отдает по одному элементу за раз; список, содержащий все шесть вариаций футболки, не создается.

В главе 17 подробно объясняется, как работают генераторы. Здесь же мы только хотели показать использование генераторных выражений для инициализации последовательностей, отличных от списков, а также для вывода последовательности, не хранящейся целиком в памяти.

Перейдем теперь к следующему фундаментальному типу последовательностей в Python: кортежу.

Кортеж – не просто неизменяемый список

В некоторых учебниках Python начального уровня кортежи описываются как «неизменяемые списки», но это описание неполно. У кортежей две функции: использование в качестве неизменяемых списков и в качестве записей с неименованными полями. Второе применение иногда незаслуженно игнорируется, поэтому начнем с него.

Кортежи как записи

В кортеже хранится запись: каждый элемент кортежа содержит данные одного поля, а его позиция определяет семантику поля.

Если рассматривать кортеж только как неизменяемый список, то количество и порядок элементов могут быть важны или не важны в зависимости от контекста. Но если считать кортеж набором полей, то количество элементов часто фиксировано, а порядок всегда важен.

В примере 2.7 показано использование кортежей в качестве записей. Отметим, что во всех случаях переупорядочение кортежа уничтожило бы информацию, потому что семантика каждого элемента данных определяется его позицией.

Пример 2.7. Кортежи как записи

```
>>> lax_coordinates = (33.9425, -118.408056) ❶
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32_450, 0.66, 8014) ❷
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ❸
...     ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids): ❹
...     print('%s/%s' % passport) ❺
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids: ❻
...     print(country)
...
USA
BRA
ESP
```

❶ Широта и долгота международного аэропорта Лос-Анджелеса.
❷ Данные о Токио: название, год, численность населения (в миллионах человек), динамика численности населения (в процентах), площадь (в км^2).
❸ Список кортежей вида (`код_страны, номер_паспорта`).
❹ При обходе списка с каждым кортежем связывается переменная `passport`.
❺ Оператор форматирования `%` понимает кортежи и трактует каждый элемент как отдельное поле.
❻ Цикл `for` знает, как извлекать элементы кортежа по отдельности, это называется «распаковкой». В данном случае второй элемент нас не интересует, поэтому он присваивается фиктивной переменной `_`.

Вообще говоря, использование `_` в качестве фиктивной переменной – не более чем удобство. Это вполне допустимое имя переменной, пусть и странное. Однако в предложении `match/case` символ `_` является метасимволом, который соответствует любому значению, но не привязан ни к какому. См. раздел «Сопоставление с последовательностью-образцом». А на консоли Python результат только что выполненной команды присваивается переменной `_`, если только он не равен `None`.

Мы часто рассматриваем записи как структуры данных с именованными полями. В главе 5 показаны два способа создания кортежей с именованными полями.

Но зачастую нет необходимости создавать класс только для того, чтобы поименовать поля, особенно если мы применяем распаковку и не используем индексы для доступа к полям. В примере 2.7 мы в одном предложении присвоили кортеж `('Tokyo', 2003, 32450, 0.66, 8014)` совокупности переменных `city`, `year`, `pop`, `chg`, `area`. Затем оператор `%` присвоил каждый элемент кортежа `passport` соответствующему спецификатору в форматной строке, переданной функции `print`. То и другое – примеры *распаковки кортежа*.

Термин *распаковка кортежа* питонисты употребляют часто, но все большее распространение получает термин *распаковка итерируемого объекта*, как, например, в заголовке документа PEP 3132 «Extended Iterable Unpacking» (*https://peps.python.org/pep-3132/*).

В разделе «Распаковка последовательностей и итерируемых объектов» сказано гораздо больше о распаковке не только кортежей, но и вообще последовательностей и итерируемых объектов.

Теперь перейдем к рассмотрению класса `tuple` как неизменяемого варианта класса `list`.

Кортежи как неизменяемые списки

Интерпретатор Python и стандартная библиотека широко используют кортежи в роли неизменяемых списков, и вам стоит последовать их примеру. У такого использования есть два важных преимущества:

Ясность

Видя в коде кортеж, мы точно знаем, что его длина никогда не изменится.

Производительность

Кортеж потребляет меньше памяти, чем список той же длины, и позволяет интерпретатору Python выполнить некоторые оптимизации.

Однако не забывайте, что неизменность кортежа относится только к хранящимся в нем ссылкам – их нельзя ни удалить, ни изменить. Но если какая-то ссылка указывает на изменяемый объект и этот объект будет изменен, то значение кортежа изменится. В следующем фрагменте иллюстрируется, что при этом происходит. Первоначально два кортежа, `a` и `b`, равны, и на рис. 2.4 показано размещение кортежа `b` в памяти.

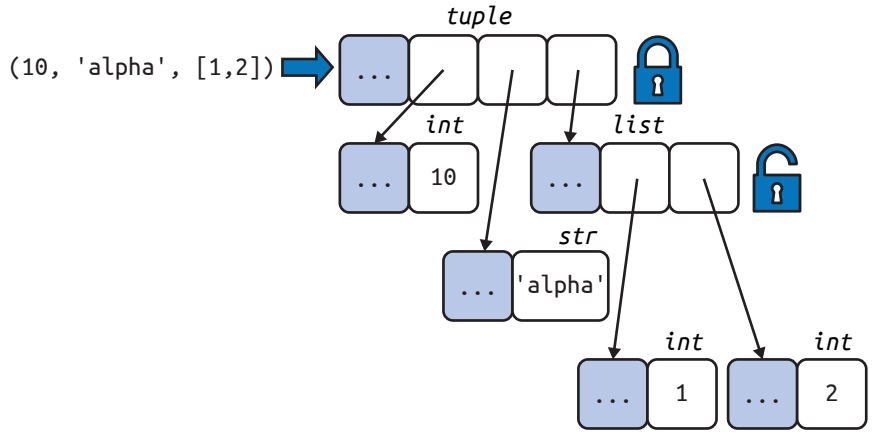

Рис. 2.4. Само содержимое кортежа неизменно, но это лишь означает, что хранящиеся в кортеже ссылки всегда указывают на одни и те же объекты. Однако если какой-то из этих объектов изменяемый, например является списком, то его содержимое может измениться

Когда последний элемент `b` изменяется, `b` и `a` становятся различны:

```
>>> a = (10, 'alpha', [1, 2])
>>> b = (10, 'alpha', [1, 2])
>>> a == b
True
>>> b[-1].append(99)
>>> a == b
False
>>> b
(10, 'alpha', [1, 2, 99])
```

Кортежи с изменяемыми элементами могут быть источником ошибок. В разделе «Что можно хешировать» мы увидим, что объект допускает хеширование только тогда, когда его значение никогда не изменяется. Нехешируемый кортеж не может быть ни ключом словаря `dict`, ни элементом множества `set`.

Если вы хотите явно узнать, является ли значение кортежа (или вообще любого объекта) фиксированным, можете воспользоваться встроенной функцией `hash` для создания функции `fixed` вида:

```
>>> def fixed(o):
...     try:
...         hash(o)
...     except TypeError:
...         return False
...     return True
...
>>> tf = (10, 'alpha', (1, 2))
>>> tm = (10, 'alpha', [1, 2])
>>> fixed(tf)
True
>>> fixed(tm)
False
```

Мы еще вернемся к этому вопросу в разделе «Относительная неизменяемость кортежей».

Несмотря на этот подвох, кортежи широко используются в качестве неизменяемых списков. Их преимущества в части производительности объяснил разработчик ядра Python Раймонд Хэттингер, отвечая на следующий вопрос, заданный на сайте StackOverflow: «Правда ли, что в Python кортежи эффективнее списков?» (*https://stackoverflow.com/questions/68630/are-tuples-more-efficient-than-lists-in-python/22140115#22140115*). Вот краткое изложение его ответа.

- Чтобы вычислить кортежный литерал, компилятор Python генерирует байт-код для константы типа кортежа, состоящий из одной операции, а для спискового литерала сгенерированный байт-код сначала помещает каждый элемент в стек данных в виде отдельной константы, а затем строит список.
- Имея кортеж `t`, вызов `tuple(t)` просто возвращает ссылку на тот же `t`. Никакого копирования не производится. Напротив, если дан список `l`, то конструктор `list(l)` должен создать новую копию `l`.
- Благодаря фиксированной длине для экземпляра `tuple` выделяется ровно столько памяти, сколько необходимо. С другой стороны, для экземпляров `list` память выделяется с запасом, чтобы амортизировать стоимость последующих добавлений в список.
- Ссылки на элементы кортежа хранятся в массиве, находящемся в самой структуре кортежа, тогда как в случае списка хранится указатель на массив ссылок, размещенный где-то в другом месте. Косвенность необходима, потому что когда список перестает помещаться в выделенной памяти, Python должен перераспределить память для массива ссылок, добавив места. Дополнительный уровень косвенности снижает эффективность процессорных кешей.

Сравнение методов кортежа и списка

При использовании типа `tuple` в качестве неизменяемого варианта типа `list` полезно знать, насколько они похожи. Из табл. 2.1 видно, что `tuple` поддерживает все методы `list`, не связанные с добавлением или удалением элементов, за одним исключением – у кортежа нет метода `__reversed__`. Но это просто оптимизация; вызов `reversed(my_tuple)` работает и без него.

Таблица 2.1. Методы и атрибуты списка и кортежа (для краткости методы, унаследованные от object, опущены)

	list	tuple	
`s.__add__(s2)`	●	●	`s + s2` – конкатенация
`s.__iadd__(s2)`	●		`s += s2` – конкатенация на месте
`s.append(e)`	●		Добавление элемента в конец списка
`s.clear()`	●		Удаление всех элементов
`s.__contains__(e)`	●	●	`e` входит в `s`
`s.copy()`	●		Поверхностная копия списка
`s.count(e)`	●	●	Подсчет числа вхождений элемента

Окончание табл. 2.1

	list	tuple	
`s.__delitem__(p)`	●		Удаление элемента в позиции `p`
`s.extend(it)`	●		Добавление в конец списка элементов из итерируемого объекта `it`
`s.__getitem__(p)`	●	●	`s[p]` – получение элемента в указанной позиции
`s.__getnewargs__()`		●	Для поддержки оптимизированной сериализации с помощью `pickle`
`s.index(e)`	●	●	Поиск позиции первого вхождения `e`
`s.insert(p, e)`	●		Вставка элемента `e` перед элементом в позиции `p`
`s.__iter__()`	●	●	Получение итератора
`s.__len__()`	●	●	`len(s)` – количество элементов
`s.__mul__(n)`	●	●	`s * n` – кратная конкатенация
`s.__imul__(n)`	●		`s *= n` – кратная конкатенация на месте
`s.__rmul__(n)`	●	●	`n * s` – инверсная кратная конкатенация[a]
`s.pop([p])`	●		Удалить и вернуть последний элемент или элемент в позиции `p`, если она задана
`s.remove(e)`	●		Удалить первое вхождение элемента `e`, заданного своим значением
`s.reverse()`	●		Изменить порядок элементов на противоположный на месте
`s.__reversed__()`	●		Получить итератор для перебора элементов от конца к началу
`s.__setitem__(p, e)`	●		`s[p] = e` – поместить `e` в позицию `p` вместо находящегося там элемента[b]
`s.sort([key], [reverse])`	●		Отсортировать элементы на месте с факультативными аргументами `key` и `reverse`

[a] Инверсные операторы рассматриваются в главе 16.

[b] Также используется для перезаписывания последовательности. См. раздел «Присваивание срезам».

Теперь перейдем к важной для идиоматического программирования на Python теме: распаковке кортежа, списка и итерируемого объекта.

Распаковка последовательностей и итерируемых объектов

Распаковка важна, потому что позволяет избежать ненужного и чреватого ошибками использования индексов для извлечения элементов из последовательностей. Кроме того, распаковка работает, когда источником данных является любой итерируемый объект, включая итераторы, которые вообще не под-

держивают индексной нотации ([]). Единственное требование – итерируемый объект должен отдавать лишь один элемент на каждую переменную на принимающей стороне, если только не используется звездочка (*) для получения всех лишних элементов (см. раздел «Использование * для выборки лишних элементов»).

Самая очевидная форма распаковки кортежа – *параллельное присваивание*, т. е. присваивание элементов итерируемого объекта кортежу переменных, как показано в следующем примере:

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates # распаковка
>>> latitude
33.9425
>>> longitude
-118.408056
```

Элегантное применение распаковки кортежа – обмен значений двух переменных без создания временной переменной:

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

Другой пример – звездочка перед аргументом при вызове функции:

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

Здесь также показано еще одно применение распаковки кортежа: возврат нескольких значений из функции способом, удобным вызывающей программе. Например, функция `os.path.split()` строит кортеж `(path, last_part)` из пути в файловой системе:

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/id_rsa.pub')
>>> filename
'id_rsa.pub'
```

Еще один способ извлечь только некоторые элементы распаковываемого кортежа – воспользоваться символом *, как описано ниже.

Использование * для выборки лишних элементов

Определение параметров функции с помощью конструкции `*args`, позволяющей получить произвольные дополнительные аргументы, – классическая возможность Python.

В Python 3 эта идея была распространена на параллельное присваивание:

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
```

```
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

В этом контексте префикс * можно поставить только перед одной переменной, которая, впрочем, может занимать любую позицию:

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

Распаковка с помощью * в вызовах функций и литеральных последовательностях

В документе PEP 448 «Additional Unpacking Generalizations» (*https://peps.python.org/pep-0448/*) предложен более гибкий синтаксис распаковки итерируемого объекта, который лучше всего описан в главе «Что нового в Python 3.5» официальной документации (*https://docs.python.org/3/whatsnew/3.5.html#pep-0448-additional-unpacking-generalizations*).

В вызовах функций можно использовать * несколько раз:

```
>>> def fun(a, b, c, d, *rest):
...     return a, b, c, d, rest
...
>>> fun(*[1, 2], 3, *range(4, 7))
(1, 2, 3, 4, (5, 6))
```

Символ * можно также использовать при определении литералов типа `list`, `tuple` и `set`, как показано в следующих примерах, взятых из официальной документации:

```
>>> *range(4), 4
(0, 1, 2, 3, 4)
>>> [*range(4), 4]
[0, 1, 2, 3, 4]
>>> {*range(4), 4, *(5, 6, 7)}
{0, 1, 2, 3, 4, 5, 6, 7}
```

В PEP 448 введен аналогичный синтаксис для оператора **, с которым мы познакомимся в разделе «Распаковка отображений».

Наконец, очень полезным свойством распаковки кортежа является возможность работы с вложенными структурами.

Распаковка вложенных объектов

Объект, в который распаковывается выражение, может содержать вложенные объекты, например `(a, b, (c, d))`, и Python правильно заполнит их, если значение имеет такую же структуру вложенности. В примере 2.8 показана распаковка вложенного объекта в действии.

Пример 2.8. Распаковка вложенных кортежей для доступа к долготе

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)), ❶
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]
def main():
    print(f'{"":15} | {"latitude":>9} | {"longitude":>9}')
    for name, _, _, (lat, lon) in metro_areas: ❷
        if lon <= 0: ❸
            print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
if __name__ == '__main__':
    main()
```

❶ Каждый кортеж содержит четыре поля, причем последнее – пара координат.
❷ Присваивая последнее поле кортежу, мы распаковываем координаты.
❸ Условие `if longitude <= 0:` отбирает только мегаполисы в Западном полушарии.

Вот что печатает эта программа:

```
                |  lat.    |  long.
Mexico City     |  19.4333 | -99.1333
New York-Newark |  40.8086 | -74.0204
Sao Paulo       | -23.5478 | -46.6358
```

Распаковку можно производить и в список, но это редко имеет смысл. Вот единственный известный мне пример, когда такая операция полезна: если запрос к базе данных возвращает ровно одну запись (например, когда в SQL-коде присутствует фраза `LIMIT 1`), то можно произвести распаковку и одновременно убедиться, что действительно возвращена одна запись:

```
>>> [record] = query_returning_single_row()
```

Если запись содержит только одно поле, то его можно получить сразу:

```
>>> [[field]] = query_returning_single_row_with_single_field()
```

Оба предложения можно было бы записать с помощью кортежей, но не забывайте об одной синтаксической тонкости: при записи одноэлементных кортежей нужно добавлять в конце запятую. Поэтому в первом случае в левой части присваивания нужно написать `(record,)`, а во втором – `((field,),)`. В обоих случаях отсутствие запятой приводит к ошибке, о которой ничего не сообщается[1].

Теперь займемся сопоставлением с образцом – операцией, которая поддерживает еще более мощные способы распаковки последовательностей.

Сопоставление с последовательностями-образцами

Самая заметная новая возможность в Python 3.10 – предложение `match/case` для сопоставления с образцом, описанное в документе PEP 634 «Structural Pattern Matching: Specification» (*https://peps.python.org/pep-0634/*).

[1] Спасибо рецензенту Леонардо Рохаэлю за этот пример.

Разработчик ядра Python Кэрол Уиллинг написал прекрасное введение в механизм сопоставления с образцом в разделе «Структурное сопоставление с образцом» главы «Что нового в Python 3.10» (*https://docs.python.org/3.10/whatsnew/3.10.html*) официальной документации. Возможно, вам захочется прочитать этот краткий обзор. Я же решил разбить тему сопоставления с образцом на части и поместить их в разные главы в зависимости от типа образца: «Сопоставление с отображением-образцом» и «Сопоставление с экземпляром класса – образцом». Развернутый пример приведен в разделе «Сопоставление с образцом в lis.py: пример».

Ниже приведен первый пример предложения `match/case`. Допустим, что мы проектируем робота, который принимает команды в виде последовательностей слов и чисел, например `BEEPER 440 3`. Разбив команду на части и разобрав числа, мы должны получить сообщение вида `['BEEPER', 440, 3]`. Для обработки таких сообщений можно воспользоваться показанным ниже методом.

Пример 2.9. Метод из гипотетического класса `Robot`

```
def handle_command(self, message):
    match message: ❶
        case ['BEEPER', frequency, times]: ❷
            self.beep(times, frequency)
        case ['NECK', angle]: ❸
            self.rotate_neck(angle)
        case ['LED', ident, intensity]: ❹
            self.leds[ident].set_brightness(ident, intensity)
        case ['LED', ident, red, green, blue]: ❺
            self.leds[ident].set_color(ident, red, green, blue)
        case _: ❻
            raise InvalidCommand(message)
```

❶ Выражение после ключевого слова `match` называется субъектом. Это данные, которые Python попытается сопоставить с образцами в ветвях `case`.

❷ С этим образцом сопоставляется любой субъект, являющийся последовательностью из трех элементов. Первый элемент должен быть равен `'BEEPER'`. Второй и третий могут быть любыми, они связываются с переменными `frequency` и `times` именно в таком порядке.

❸ С этим образцом сопоставляется любой субъект, содержащий два элемента, причем первый должен быть равен `'NECK'`.

❹ С этим образцом сопоставляется субъект, содержащий три элемента, первым из которых должен быть `'LED'`. Если число элементов не совпадает, то Python переходит к следующей ветви `case`.

❺ Еще одна последовательность-образец, начинающаяся с `'LED'`, но теперь содержащая пять элементов, включая константу `'LED'`.

❻ Это ветвь `case` по умолчанию. С ней сопоставляется любой субъект, для которого не нашлось подходящего образца. Переменная `_` специальная, как мы увидим ниже.

На первый взгляд, конструкция `match/case` похожа на предложение `switch/case` в языке C – но это только на первый взгляд[1]. Основное улучшение `match` по сравнению с `switch` – деструктуризация, т. е. более развитая форма распаковки. Деструктуризация – новое слово в словаре Python, но оно часто встречается в документации по языкам, поддерживающим сопоставление с образцом, например Scala и Elixir.

Для начала в примере 2.10 показана часть примера 2.8, переписанная с использованием `match/case`.

Пример 2.10. Деструктуризация вложенных кортежей (необходима версия Python ≥ 3.10)

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]
def main():
    print(f'{"":15} | {"latitude":>9} | {"longitude":>9}')
    for record in metro_areas:
        match record: ❶
            case [name, _, _, (lat, lon)] if lon <= 0: ❷
                print(f'{name:15} | {lat: 9.4f} | {lon: 9.4f}')
```

❶ Субъектом в этом предложении `match` является `record`, т. е. каждый из кортежей в списке `metro_areas`.

❷ Ветвь `case` состоит из двух частей: образец и необязательное охранное условие, начинающееся ключевым словом `if`.

В общем случае сопоставление с последовательностью-образцом считается успешным, если:

1) субъект является последовательностью *и*;
2) субъект и образец содержат одинаковое число элементов *и*;
3) все соответственные элементы, включая вложенные, совпадают.

Например, образец `[name, _, _, (lat, lon)]` в примере 2.10 сопоставляется с последовательностью из четырех элементов, последним элементом которой является последовательность из двух элементов.

Последовательности-образцы могут быть кортежами, списками или любой комбинацией вложенных кортежей и списков, на синтаксисе это никак не сказывается: квадратные и круглые скобки в последовательности-образце означают одно и то же. Я записал образец в виде списка с вложенным 2-кортежем, просто чтобы избежать повторения квадратных или круглых скобок, как в примере 2.10.

[1] На мой взгляд, последовательность блоков `if/elif/elif/.../else` является достойной заменой `switch/case`. Она исключает такие проблемы, как случайное проваливание (*https://en.wikipedia.org/wiki/Switch_statement#Fallthrough*) и висячие ветви `else` (*https://en.wikipedia.org/wiki/Dangling_else*), которые проектировщики некоторых языков некритически скопировали из C, – через много лет было признано, что они являются источником бесчисленных ошибок.

Последовательность-образец может сопоставляться с большинством реальных или виртуальных подклассов класса `collections.abc.Sequence`, за исключением лишь классов `str`, `bytes` и `bytearray`.

Экземпляры классов `str`, `bytes` и `bytearray` не считаются последовательностями в контексте `match/case`. Субъект `match`, принадлежащий одному из этих типов, трактуется как «атомарное» значение – точно так же, как целое число 987 считается одним значением, а не последовательностью цифр. Обращение с этими типами как с последовательностями могло бы стать причиной ошибок из-за непреднамеренного совпадения. Если вы хотите рассматривать объект одного из этих типов как субъект последовательности, то преобразуйте его тип во фразе `match`. Например, так мы поступили с `tuple(phone)` в следующем фрагменте:

```
match tuple(phone):
    case ['1', *rest]: # Северная Америка и страны Карибского бассейна
        ...
    case ['2', *rest]: # Африка и некоторые другие территории
        ...
    case ['3' | '4', *rest]: # Европа
        ...
```

С последовательностями-образцами совместимы следующие типы из стандартной библиотеки:

```
list     memoryview   array.array
tuple    range        collections.deque
```

В отличие от распаковки, образцы не деструктурируют итерируемые объекты, не являющиеся последовательностями (например, итераторы).

Символ `_` в образцах имеет специальный смысл: он сопоставляется с одним любым элементом в этой позиции, но никогда не связывается со значением сопоставленного элемента. Кроме того, `_` – единственная переменная, которая может встречаться в образце более одного раза.

Любую часть образца можно связать с переменной с помощью ключевого слова `as`:

```
case [name, _, _, (lat, lon) as coord]:
```

Субъект `['Shanghai', 'CN', 24.9, (31.1, 121.3)]` сопоставляется с этим образцом, и при этом устанавливаются следующие переменные:

Переменная	Установленное значение
`name`	`'Shanghai'`
`lat`	`31.1`
`lon`	`121.3`
`coord`	`(31.1, 121.3)`

Образцы можно сделать более специфичными, добавив информацию о типе. Например, показанный ниже образец сопоставляется с последовательностью с такой же структурой вложенности, как в предыдущем примере, но первый

элемент должен быть экземпляром типа `str` и оба элемента 2-кортежа должны иметь тип `float`:

```
case [str(name), _, _, (float(lat), float(lon))]:
```

Выражения `str(name)` и `float(lat)` выглядят как вызовы конструкторов, как было бы, если бы мы хотели преобразовать `name` и `lat` соответственно в типы `str` и `float`. Но в контексте образца эта синтаксическая конструкция производит проверку типа во время выполнения: образец сопоставится с 4-элементной последовательностью, в которой элемент 0 должен иметь тип `str`, а элемент 3 должен быть парой чисел типа `float`. Кроме того, `str` в позиции 0 будет связана с переменной `name`, а два числа типа `float` в позиции 3 – с переменными `lat` и `lon` соответственно. Таким образом, хотя `str(name)` заимствует синтаксис конструктора, в контексте образца семантика совершенно другая. Использование произвольных классов в образцах рассматривается в разделе «Сопоставление с экземплярами классов – образцами».

С другой стороны, если мы хотим произвести сопоставление произвольной последовательности-субъекта, начинающейся с `str` и заканчивающейся вложенной последовательностью из двух `float`, то можем написать:

```
case [str(name), *_, (float(lat), float(lon))]:
```

Здесь `*_` сопоставляется с любым числом элементов без привязки их к переменной. Если вместо `*_` использовать `*extra`, то с переменной `extra` будет связан список `list`, содержащий 0 или более элементов.

Необязательное охранное условие, начинающееся со слова `if`, вычисляется только в случае успешного сопоставления с образцом. При этом в условии можно ссылаться на переменные, встречающиеся в образце, как в примере 2.10:

```
match record:
    case [name, _, _, (lat, lon)] if lon <= 0:
        print(f'{name:15} | {lat: 9.4f} | {lon: 9.4f}')
```

Вложенный блок, содержащий предложение `print`, выполняется, только если сопоставление было успешным и охранное условие *похоже на истину*.

Деструктуризация с помощью образцов настолько выразительна, что иногда даже наличие единственной ветви `case` может сделать код проще. Гвидо ван Россум собрал коллекцию примеров `case/match`, один из которых назвал «Очень глубокий итерируемый объект и сравнение типа с выделением» (*https://github.com/gvanrossum/ patma/blob/3ece6444ef70122876fd9f0099eb9490a2d630df/EXAMPLES.md#case-6-a-very-deep-iterable-and-type-match-with-extraction*).

Нельзя сказать, что пример 2.10 чем-то лучше примера 2.8. Это просто два разных способа сделать одно и то же. В следующем примере мы покажем, как сопоставление с образцом позволяет писать более ясный, лаконичный и эффективный код.

Сопоставление с последовательностями-образцами в интерпретаторе

Питер Норвиг из Стэнфордского университета написал программу *lis.py*, интерпретатор подмножества диалекта Scheme языка программирования Lisp. Она состоит всего из 132 строк красивого и прекрасно читаемого кода на Python. Я взял код Норвига, распространяемый по лицензии MIT, и перенес его на Python 3.10, чтобы продемонстрировать сопоставление с образцом. В этом разделе мы сравним ключевую часть кода Норвига – ту, в которой используется `if/elif` и распаковка, – с вариантом на основе `match/case`.

Две главные функции в *lis.py* – `parse` и `evaluate`[1]. Анализатор принимает выражение Scheme со скобками и возвращает списки Python. Приведем два примера:

```
>>> parse('(gcd 18 45)')
['gcd', 18, 45]
>>> parse('''
... (define double
...     (lambda (n)
...         (* n 2)))
... ''')
['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]
```

Вычислитель принимает такого рода списки и выполняет их. В первом примере вызывается функция `gcd` с аргументами `18` и `45`. Она вычисляет наибольший общий делитель аргументов: 9. Во втором примере определена функция `double` с параметром `n`. Ее телом является выражение `(* n 2)`. Результат вызова этой функции в Scheme – значение последнего выражения в ее теле.

Нас здесь больше всего интересует деструктуризация последовательностей, поэтому я не стану вдаваться в действия вычислителя. Подробнее о работе *lis.py* можно прочитать в разделе «Сопоставление с образцом в lis.py: пример».

В примере 2.11 показан слегка модифицированный вычислитель Норвига, в котором я оставил только код для демонстрации последовательностей-образцов.

Пример 2.11. Сопоставление с образцами без `match/case`

```
def evaluate(exp: Expression, env: Environment) -> Any:
    "Evaluate an expression in an environment."
    if isinstance(exp, Symbol): # ссылка на переменную
        return env[exp]
    # ... несколько строк опущено
    elif exp[0] == 'quote':     # (quote exp)
        (_, x) = exp
        return x
    elif exp[0] == 'if':        # (if test conseq alt)
        (_, test, consequence, alternative) = exp
        if evaluate(test, env):
            return evaluate(consequence, env)
        else:
            return evaluate(alternative, env)
```

[1] Последняя называется `eval` в коде Норвига; я переименовал ее, чтобы избежать путаницы со встроенной в Python `eval`.

```
    elif exp[0] == 'lambda':    # (lambda (parm…) body…)
        (_, parms, *body) = exp
        return Procedure(parms, body, env)
    elif exp[0] == 'define':
        (_, name, value_exp) = exp
        env[name] = evaluate(value_exp, env)
    # ... последующие строки опущены
```

Обратите внимание, что в каждой ветви `elif` проверяется первый элемент списка, а затем список распаковывается и первый элемент игнорируется. Столь активное использование распаковки наводит на мысль, что Норвиг – большой поклонник сопоставления с образцом, но этот код был написан для Python 2 (хотя работает и с любой версией Python 3).

Воспользовавшись предложением `match/case` в Python ⩾ 3.10, мы сможем переписать `evaluate`, как показано в примере 2.12.

Пример 2.12. Сопоставление с образцом с применением `match/case` (необходима версия Python ⩾ 3.10)

```
def evaluate(exp: Expression, env: Environment) -> Any:
    "Evaluate an expression in an environment."
    match exp:
    # ... несколько строк опущено
        case ['quote', x]: ❶
            return x
        case ['if', test, consequence, alternative]: ❷
            if evaluate(test, env):
                return evaluate(consequence, env)
            else:
                return evaluate(alternative, env)
        case ['lambda', [*parms], *body] if body: ❸
            return Procedure(parms, body, env)
        case ['define', Symbol() as name, value_exp]: ❹
            env[name] = evaluate(value_exp, env)
        # ... еще несколько строк опущено
        case _: ❺
            raise SyntaxError(lispstr(exp))
```

❶ Сопоставляется, если субъект – двухэлементная последовательность, начинающаяся с `'quote'`.

❷ Сопоставляется, если субъект – четырехэлементная последовательность, начинающаяся с `'if'`.

❸ Сопоставляется, если субъект – последовательность из трех или более элементов, начинающаяся с `'lambda'`. Охранное условие гарантирует, что `body` не пусто.

❹ Сопоставляется, если субъект – трехэлементная последовательность, начинающаяся с `'define'`.

❺ Рекомендуется всегда включать перехватывающую ветвь `case`. В данном случае если `exp` не сопоставляется ни с одним образцом, значит, выражение построено неправильно, и я возбуждаю исключение `SyntaxError`.

Если бы перехватывающей ветви не было, то предложение ничего не сделало бы, когда субъект не сопоставляется ни с одной ветвью `case`, т. е. имела бы место ошибка без каких бы то ни было сообщений.

Норвиг сознательно опустил проверку ошибок в *lis.py*, чтобы сделать код понятнее. Сопоставление с образцом позволяет добавить больше проверок, сохранив удобочитаемость. Например, в образце `'define'` оригинальный код не проверяет, что `name` является экземпляром класса `Symbol`, потому что это потребовало бы включения блока `if`, вызова `isinstance` и дополнительного кода. Пример 2.12 короче и безопаснее, чем пример 2.11.

Альтернативные образцы для лямбда-выражений

В языке Scheme имеется синтаксическая конструкция `lambda`, в ней используется соглашение о том, что суффикс `…` означает, что элемент может встречаться нуль или более раз:

```
(lambda (parms…) body1 body2…)
```

Простой образец для сопоставления с `'lambda'` мог бы выглядеть так:

```
case ['lambda', parms, *body] if body:
```

Однако с ним сопоставляется любое значение в позиции `parms`, в частности первое `'x'` в следующем недопустимом субъекте:

```
['lambda', 'x', ['*', 'x', 2]]
```

В Scheme во вложенном списке после ключевого слова `lambda` находятся имена формальных параметров функции, и это должен быть именно список, даже если он содержит всего один элемент. Список может быть и пустым, если функция не имеет параметров, как `random.random()` в Python.

В примере 2.12 я сделал образец `'lambda'` более безопасным, воспользовавшись вложенной последовательностью-образцом:

```
case ['lambda', [*parms], *body] if body:
    return Procedure(parms, body, env)
```

В каждой последовательности-образце `*` может встречаться только один раз. Здесь же мы имеем две последовательности: внешнюю и внутреннюю.

Добавление символов `[*]` вокруг `parms` сделало образец более похожим на синтаксическую конструкцию Scheme, которую он призван обрабатывать, так что мы получаем дополнительную проверку правильности структуры.

Сокращенный синтаксис для определения функции

В Scheme имеется альтернативная синтаксическая конструкция `define` для создания именованной функции без использования вложенного `lambda`, а именно:

```
(define (name parm…) body1 body2…)
```

За ключевым словом `define` должен следовать список, содержащий имя `name` новой функции и нуль или более имен параметров. После этого списка располагается тело функции, содержащее одно или несколько выражений.

Добавив в `match` следующие две строки, мы обработаем этот случай:

```
case ['define', [Symbol() as name, *parms], *body] if body:
  env[name] = Procedure(parms, body, env)
```

Я бы поместил эту ветвь `case` после другой ветви, сопоставляющей с `define` в примере 2.12. В данном случае порядок расположения двух ветвей неважен,

потому что никакой субъект не может сопоставляться сразу с обоими образцами. Действительно, в первоначальной ветви `define` второй элемент должен иметь тип `Symbol`, а в ветви для определения функции он должен быть последовательностью, начинающейся с `Symbol`.

А теперь подумайте, сколько кода пришлось бы добавить, чтобы поддержать вторую синтаксическую конструкцию, включающую `define`, не прибегая к сопоставлению с образцом (пример 2.11). Предложение `match` делает куда больше, чем `switch` в C-подобных языках.

Сопоставление с образцом – пример декларативного программирования: мы описываем, «что» хотим сопоставить, а не «как» это сделать. Форма кода повторяет форму данных, как видно из табл. 2.2.

Таблица 2.2. Некоторые синтаксические конструкции Scheme и соответствующие им образцы

Конструкция Scheme	Последовательность-образец
`(quote exp)`	`['quote', exp]`
`(if test conseq alt)`	`['if', test, conseq, alt]`
`(lambda (parms…) body1 body2…)`	`['lambda', [*parms], *body] if body`
`(define name exp)`	`['define', Symbol() as name, exp]`
`(define (name parms…) body1 body2…)`	`['define', [Symbol() as name, *parms], *body] if body`

Надеюсь, эта переработка функции `evaluate` в коде Норвига с применением сопоставления с образцом убедила вас в том, что предложение `match/case` может сделать ваш код более понятным и безопасным.

Мы еще вернемся к программе *lis.py* в разделе «Сопоставление с образцом в lis.py: пример», где рассмотрим полный код `match/case` в функции `evaluate`. Если хотите больше узнать о программе Норвига *lis.py*, прочитайте его замечательную статью «How to Write a (Lisp) Interpreter (in Python))» (*https://norvig.com/lispy.html*).

На этом завершается наше первое знакомство с распаковкой, деструктуризацией и сопоставлением с последовательностями-образцами. Другие типы образцов будут рассмотрены в последующих главах.

Каждый пишущий на Python программист знает о синтаксисе вырезания частей последовательности – `s[a:b]`. А мы сейчас рассмотрим менее известные факты об операции получения среза.

Получение среза

Общей особенностью классов `list`, `tuple`, `str` и прочих типов последовательностей в Python является поддержка операций среза, которые обладают куда большими возможностями, чем многие думают.

В этом разделе мы опишем *использование* дополнительных форм срезки. А о том, как реализовать их в пользовательских классах, поговорим в главе 12, не отступая от общей установки – в этой части рассматривать готовые классы, а в части III – создание новых.

Почему в срезы и диапазоны не включается последний элемент

Принятое в Python соглашение не включать последний элемент в срезы и диапазоны соответствует индексации с нуля, принятой в Python, C и многих других языках. Приведем несколько полезных следствий из этого соглашения.

- Легко понять, какова длина среза или диапазона, если задана только конечная позиция: и `range(3)`, и `my_list[:3]` содержат три элемента.
- Легко вычислить длину среза или диапазона, если заданы начальная и конечная позиции, достаточно вычислить их разность `stop - start`.
- Легко разбить последовательность на две непересекающиеся части по любому индексу `x`: нужно просто взять `my_list[:x]` и `my_list[x:]`. Например:

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2] # split at 2
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3] # split at 3
[10, 20, 30]
>>> l[3:]
[40, 50, 60]
```

Но самые убедительные аргументы в пользу этого соглашения изложил голландский ученый, специализирующийся в информатике, Эдсгер Вибе Дейкстра (см. последний пункт в списке дополнительной литературы).

Теперь познакомимся ближе с тем, как Python интерпретирует нотацию среза.

Объекты среза

Хотя это не секрет, все же напомним, что в выражении `s[a:b:c]` задается шаг `c`, что позволяет вырезать элементы не подряд. Шаг может быть отрицательным, тогда элементы вырезаются от конца к началу. Поясним на примерах:

```
>>> s = 'bicycle'
>>> s[::3]
'bye'
>>> s[::-1]
'elcycib'
>>> s[::-2]
'eccb'
```

Еще один пример был приведен в главе 1, где мы использовали выражение `deck[12::13]` для выборки всех тузов из неперетасованной колоды:

```
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Нотация `a:b:c` допустима только внутри квадратных скобок, когда используется в качестве оператора индексирования и порождает объект среза `slice(a, b, c)`. В разделе «Как работает срезка» главы 12 мы увидим, что для вычисления выражения `seq[start:stop:step]` Python вызывает метод `seq.__getitem__(slice(start, stop, step))`. Даже если вы никогда не будете сами реализовывать типы после-

довательностей, знать об объектах среза полезно, потому что это позволяет присваивать срезам имена – по аналогии с именами диапазонов ячеек в электронных таблицах.

Пусть требуется разобрать плоский файл данных, например накладную, показанную в примере 2.13. Вместо того чтобы загромождать код «зашитыми» диапазонами, мы можем поименовать их. Посмотрим, насколько понятным становится при этом цикл `for` в конце примера.

Пример 2.13. Строки из файла накладной

```
>>> invoice = """
...
0.....6.................................40........52...55........
... 1909  Pimoroni PiBrella                     $17.50 3   $52.50
... 1489  6mm Tactile Switch x20                 $4.95 2    $9.90
... 1510  Panavise Jr. - PV-201                 $28.00 1   $28.00
... 1601  PiTFT Mini Kit 320x240                $34.95 1   $34.95
... """
>>> SKU = slice(0, 6)
>>> DESCRIPTION = slice(6, 40)
>>> UNIT_PRICE = slice(40, 52)
>>> QUANTITY = slice(52, 55)
>>> ITEM_TOTAL = slice(55, None)
>>> line_items = invoice.split('\n')[2:]
>>> for item in line_items:
... print(item[UNIT_PRICE], item[DESCRIPTION])
...
    $17.50 Pimoroni PiBrella
     $4.95 6mm Tactile Switch x20
    $28.00 Panavise Jr. - PV-201
    $34.95 PiTFT Mini Kit 320x240
```

Мы еще вернемся к объектам `slice`, когда дойдем до создания собственных коллекций в разделе «Vector, попытка № 2: последовательность, допускающая срезку» главы 12. А пока отметим, что с точки зрения пользователя у операции срезки есть ряд дополнительных возможностей, в частности многомерные срезы и нотация многоточия (...). Читайте дальше.

Многомерные срезы и многоточие

Оператор `[]` может принимать несколько индексов или срезов, разделенных запятыми. Специальные методы `__getitem__` и `__setitem__`, на которых основан оператор `[]`, просто принимают индексы, заданные в выражении `a[i, j]`, в виде кортежа. Иначе говоря, для вычисления `a[i, j]` Python вызывает `a.__getitem__((i, j))`.

Это используется, например, во внешнем пакете NumPy, где для получения одного элемента двумерного массива `numpy.ndarray` применяется нотация `a[i, j]`, а для получения двумерного среза – нотация `a[m:n, k:l]`. В примере 2.22 ниже будет продемонстрировано использование этой нотации.

За исключением `memoryview`, в Python встроены только одномерные типы последовательностей, поэтому они поддерживают лишь один индекс или срез, а не кортеж[1].

[1] В разделе «Представления памяти» будет показано, что специально сконструированные представления памяти могут иметь более одного измерения.

Многоточие – записывается в виде трех отдельных точек, а не одного символа … (Unicode U+2026) – распознается анализатором Python как лексема. Это псевдоним объекта `Ellipsis`, единственного экземпляра класса `ellipsis`[1]. А раз так, то многоточие можно передавать в качестве аргумента функциям и использовать в качестве части спецификации среза, например `f(a, ..., z)` или `a[i:...]`. В NumPy … используется для сокращенного задания среза многомерного массива; например, если `x` – четырехмерный массив, то `x[i, ...]` – то же самое, что `x[i, :, :, :,]`. Дополнительные сведения по этому вопросу можно найти в «Кратком введении в NumPy» (*https://numpy.org/doc/stable/user/quickstart.html#indexing-slicing-and-iterating*).

На момент написания этой книги мне не было известно о применении объекта `Ellipsis` или многомерных индексов в стандартной библиотеке Python. Если найдете, дайте мне знать. Эти синтаксические средства существуют для поддержки пользовательских типов и таких расширений, как NumPy.

Срезы полезны не только для выборки частей последовательности; они позволяют также модифицировать изменяемые последовательности на месте, т. е. не перестраивая с нуля.

Присваивание срезу

Изменяемую последовательность можно расширять, схлопывать и иными способами модифицировать на месте, применяя нотацию среза в левой части оператора присваивания или в качестве аргумента оператора `del`. Следующие примеры дают представление о возможностях этой нотации:

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
6
7
>>> l[2:5] = 100 ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

❶ Когда в левой части присваивания стоит срез, в правой должен находиться итерируемый объект, даже если он содержит всего один элемент.

[1] Нет, я ничего не перепутал: имя класса `ellipsis` записывается строчными буквами, а его экземпляр – встроенный объект `Ellipsis`. Точно так же обстоит дело с классом `bool` и его экземплярами `True` и `False`.

Все знают, что конкатенация – распространенная операция для последовательностей любого типа. В учебниках Python для начинающих объясняется, как использовать для этой цели операторы `+` и `*`, однако в их работе есть кое-какие тонкие детали, которые мы сейчас и обсудим.

Использование + и * для последовательностей

Пишущие на Python программисты ожидают от последовательностей поддержки операторов `+` и `*`. Обычно оба операнда `+` должны быть последовательностями одного типа, причем ни один из них не модифицируется, а создается новая последовательность того же типа, которая и является результатом конкатенации.

Для конкатенации нескольких экземпляров одной последовательности ее можно умножить на целое число. При этом также создается новая последовательность:

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

Операторы `+` и `*` всегда создают новый объект и никогда не изменяют свои операнды.

Остерегайтесь выражений вида `a * n`, где `a` – последовательность, содержащая изменяемые элементы, потому что результат может оказаться неожиданным. Например, при попытке инициализировать список списков `my_list = [[]] * 3` получится список, содержащий три ссылки на один и тот же внутренний список, хотя вы, скорее всего, хотели не этого.

В следующем разделе мы рассмотрим ловушки, которые подстерегают нас при попытке использовать `*` для инициализации списка списков.

Построение списка списков

Иногда требуется создать список, содержащий несколько вложенных списков, например чтобы распределить студентов по группам или представить клетки на игровой доске. Лучше всего это делать с помощью спискового включения, как показано в примере 2.14.

Пример 2.14. Список, содержащий три списка длины 3, может представлять поле для игры в крестики и нолики

```
>>> board = [['_'] * 3 for i in range(3)] ❶
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X' ❷
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

❶ Создать список из трех списков по три элемента в каждом. Взглянуть на его структуру.

❷ Поместить крестик в строку 1, столбец 2 и проверить, что получилось.

Соблазнительный, но ошибочный короткий путь показан в примере 2.15.

Пример 2.15. Список, содержащий три ссылки на один и тот же список, бесполезен

```
>>> weird_board = [['_'] * 3] * 3 ❶
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> weird_board[1][2] = 'O' ❷
>>> weird_board
[['_', '_', 'O'], ['_', '_', 'O'], ['_', '_', 'O']]
```

❶ Внешний список содержит три ссылки на один и тот же внутренний список. Пока не сделано никаких изменений, все кажется нормальным.

❷ Поместив нолик в строку 1, столбец 2, мы обнаруживаем, что все строки ссылаются на один и тот же объект.

Проблема в том, что код в примере 2.15, по существу, ведет себя так же, как следующий код:

```
row = ['_'] * 3
board = []
for i in range(3):
    board.append(row) ❶
```

❶ Один и тот же объект `row` трижды добавляется в список `board`.

С другой стороны, списковое включение из примера 2.14 эквивалентно такому коду:

```
>>> board = []
>>> for i in range(3):
...     row = ['_'] * 3 ❶
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board ❷
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

❶ На каждой итерации строится новый список `row`, который добавляется в конец списка `board`.

❷ Как и положено, изменилась только строка 2.

Если проблема или ее решение, представленные в этом разделе, вам не вполне понятны, не огорчайтесь. Глава 6 специально написана для того, чтобы прояснить механизм работы ссылок и изменяемых объектов, а также связанные с ним подводные камни.

До сих пор мы говорили о простых операторах `+` и `*` в применении к последовательностям, но существуют также операторы `+=` и `*=`, которые работают совершенно по-разному в зависимости от того, изменяема конечная последовательность или нет. Эти различия объяснены в следующем разделе.

Составное присваивание последовательностей

Поведение операторов составного присваивания `+=` и `*=` существенно зависит от типа первого операнда. Для простоты мы рассмотрим составное сложение (`+=`), но все сказанное равным образом относится также к оператору `*=` и другим операторам составного присваивания.

За оператором `+=` стоит специальный метод `__iadd__` (аббревиатура «in-place addition» – сложение на месте). Но если метод `__iadd__` не реализован, то Python вызывает метод `__add__`. Рассмотрим следующее простое выражение:

```
>>> a += b
```

Если объект `a` реализует метод `__iadd__`, то он и будет вызван. В случае изменяемых последовательностей (например, `list`, `bytearray`, `array.array`) `a` будет изменен на месте (результат получается такой же, как при вызове `a.extend(b)`). Если же `a` не реализует `__iadd__`, то выражение `a += b` вычисляется так же, как `a = a + b`, т. е. сначала вычисляется `a + b` и получившийся в результате новый объект связывается с переменной `a`. Иными словами, идентификатор объекта `a` остается тем же самым или становится другим в зависимости от наличия метода `__iadd__`.

Вообще говоря, если последовательность изменяемая, то можно ожидать, что метод `__iadd__` реализован и оператор `+=` выполняет сложение на месте. В случае неизменяемых последовательностей такое, очевидно, невозможно.

Сказанное об операторе `+=` применимо также к оператору `*=`, который реализован с помощью метода `__imul__`. Специальные методы `__iadd__` и `__imul__` обсуждаются в главе 16. Ниже демонстрируется применение оператора `*=` к изменяемой и неизменяемой последовательностям:

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800 ❶
>>> l *= 2
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
4311953800 ❷
>>> t = (1, 2, 3)
>>> id(t)
4312681568 ❸
>>> t *= 2
>>> id(t)
4301348296 ❹
```

❶ Идентификатор исходного списка.
❷ После умножения список – тот же самый объект, в который добавлены новые элементы.
❸ Идентификатор исходного кортежа.
❹ В результате умножения создан новый кортеж.

Кратная конкатенация неизменяемых последовательностей выполняется неэффективно, потому что вместо добавления новых элементов интерпрета-

тор вынужден копировать всю конечную последовательность, чтобы создать новую с добавленными элементами[1].

Мы рассмотрели типичные случаи использования оператора `+=`. А в следующем разделе обсудим интригующий случай, показывающий, что в действительности означает «неизменяемость» в контексте кортежей.

Головоломка: присваивание A +=

Попробуйте, не прибегая к оболочке, ответить на вопрос: что получится в результате вычисления двух выражений в примере 2.16[2]?

Пример 2.16. Загадка

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
```

Что произойдет в результате? Какой ответ кажется вам правильным?

1. `t` принимает значение `(1, 2, [30, 40, 50, 60])`.
3. Возбуждается исключение `TypeError` с сообщением о том, что объект `'tuple'` не поддерживает присваивание.
3. Ни то, ни другое.
4. И то, и другое.

Я был почти уверен, что правильный ответ **b**, но на самом деле правилен ответ **d**: «И то, и другое»! В примере 2.17 показано, как этот код выполняется в оболочке для версии Python 3.9[3].

Пример 2.17. Неожиданный результат: элемент `t2` изменился и возбуждено исключение

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, [30, 40, 50, 60])
```

Сайт Online Python Tutor (*http://www.pythontutor.com/*) – прекрасный инструмент для наглядной демонстрации работы Python. На рис. 2.5 приведены два снимка экрана, демонстрирующих начальное и конечное состояния кортежа `t` после выполнения кода из примера 2.17.

---

[1] Тип `str` – исключение из этого правила. Поскольку построение строки с помощью оператора `+=` в цикле – весьма распространенная операция, в CPython этот случай оптимизирован. Экземпляры `str` создаются с запасом памяти, чтобы при конкатенации не приходилось каждый раз копировать всю строку.

[2] Спасибо Леонардо Рохаэлю и Сезару Каваками, которые предложили эту задачу на Бразильской конференции по языку Python 2013 года.

[3] Один читатель указал, что операцию из этого примера можно без ошибок выполнить с помощью выражения `t[2].extend([50,60])`. Я это знаю, но цель примера – обсудить странное поведение оператора `+=`.

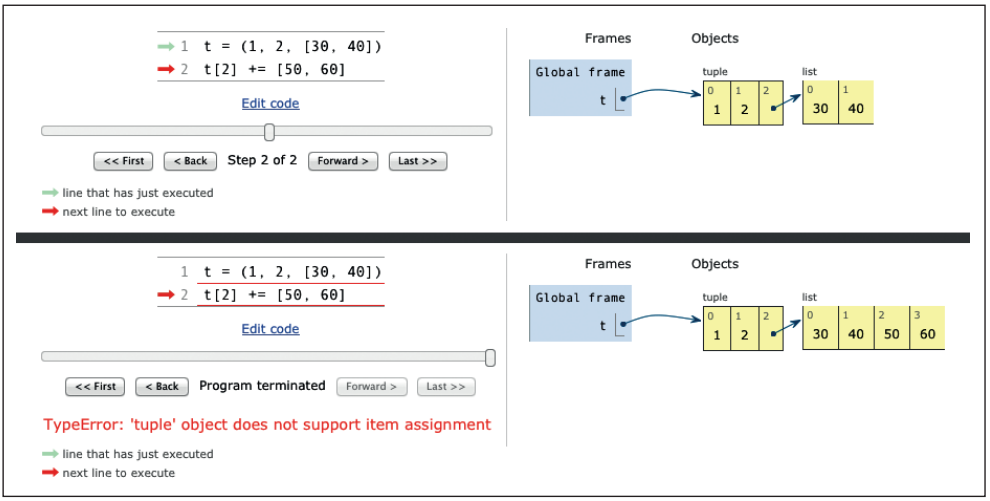

Рис. 2.5. Начальное и конечное состояния кортежа в задаче о присваивании (диаграммы сгенерированы на сайте Online Python Tutor)

Изучение байт-кода, который Python генерирует для выражения `s[a] += b` (пример 2.18), показывает, что происходит на самом деле.

Пример 2.18. Байт-код вычисления выражения s[a] += b

```
>>> dis.dis('s[a] += b')
 1           0 LOAD_NAME                0 (s)
             3 LOAD_NAME                1 (a)
             6 DUP_TOP_TWO
             7 BINARY_SUBSCR                ❶
             8 LOAD_NAME                2 (b)
            11 INPLACE_ADD                  ❷
            12 ROT_THREE
            13 STORE_SUBSCR                 ❸
            14 LOAD_CONST               0 (None)
            17 RETURN_VALUE
```

❶ Поместить значение `s[a]` на вершину стека (`TOS`).
❷ Выполнить `TOS += b`. Эта операция завершается успешно, если `TOS` ссылается на изменяемый объект (в примере 2.17 это список).
❸ Выполнить присваивание `s[a] = TOS`. Эта операция завершается неудачно, если `s` – неизменяемый объект (в примере 2.17 это кортеж `t`).

Это патологический случай – за 20 лет, что я пишу на Python, я ни разу не слышал, чтобы кто-то нарвался на такое поведение на практике.

Но из этого примера я вынес три урока.

- Не стоит помещать изменяемые элементы в кортежи.
- Составное присваивание – не атомарная операция; мы только что видели, как она возбуждает исключение, проделав часть работы.
- Изучить байт-код не так уж трудно, и часто это помогает понять, что происходит под капотом.

Познакомившись с тонкостями использования операторов `+` и `*` для конкатенации, сменим тему и обратимся еще к одной важной операции с последовательностями: сортировке.

Метод list.sort и встроенная функция sorted

Метод `list.sort` сортирует список на месте, т. е. не создавая копию. Он возвращает `None`, напоминая, что изменяет объект-приемник[1], а не создает новый список. Это важное соглашение в Python API: функции и методы, изменяющие объект на месте, должны возвращать `None`, давая вызывающей стороне понять, что изменился сам объект в противовес созданию нового. Аналогичное поведение демонстрирует, к примеру, функция `random.shuffle`, которая перетасовывает изменяемую последовательность `s` на месте и возвращает `None`.

У соглашения о возврате `None` в случае обновления на месте есть недостаток: такие методы невозможно соединить в цепочку. Напротив, методы, возвращающие новые объекты (например, все методы класса `str`), можно сцеплять, получая тем самым «текучий» интерфейс. Дополнительные сведения по этому вопросу см. в статье Википедии «Fluent interface» (*http://en.wikipedia.org/wiki/Fluent_interface*).

С другой стороны, встроенная функция `sorted` создает и возвращает новый список. На самом деле она принимает любой итерируемый объект в качестве аргумента, в том числе неизменяемые последовательности и генераторы (см. главу 147). Но независимо от типа исходного итерируемого объекта `sorted` всегда возвращает новый список.

И метод `list.sort`, и функция `sorted` принимают два необязательных именованных аргумента:

`reverse`

Если `True`, то элементы возвращаются в порядке убывания (т. е. инвертируется сравнение элементов). По умолчанию `False`.

`key`

Функция с одним аргументом, которая вызывается для каждого элемента и возвращает его ключ сортировки. Например, если при сортировке списка строк задать `key=str.lower`, то строки будут сортироваться без учета регистра, а если `key=len`, то по длине в символах. По умолчанию подразумевается тождественная функция (т. е. сравниваются сами элементы).

Необязательный именованный параметр `key` можно также использовать совместно с встроенными функциями `min()` и `max()` и другими функциями из стандартной библиотеки (например, `itertools.groupby()` или `heapq.nlargest()`).

[1] Приемником называется объект, от имени которого вызывается метод, т. е. объект, связанный с переменной `self` в теле метода.

Примеры ниже иллюстрируют применение этих функций и именованных аргументов. Они также демонстрируют, что алгоритм сортировки в Python устойчивый (т. е. сохраняет относительный порядок элементов, признанных равными)[1]:

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry'] ❶
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❷
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple'] ❸
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry'] ❹
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple'] ❺
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❻
>>> fruits.sort()                          ❼
>>> fruits
['apple', 'banana', 'grape', 'raspberry'] ❽
```

❶ Порождает новый список строк, отсортированный в алфавитном порядке[2].
❷ Инспекция исходного списка показывает, что он не изменился.
❸ Это сортировка в обратном алфавитном порядке.
❹ Новый список строк, отсортированный уже по длине. Поскольку алгоритм сортировки устойчивый, строки «grape» и «apple», обе длины 5, остались в том же порядке.
❺ Здесь строки отсортированы в порядке убывания длины. Результат не является инверсией предыдущего, потому что в силу устойчивости сортировки «grape» по-прежнему оказывается раньше «apple».
❻ До сих пор порядок исходного списка `fruits` не изменился.
❼ Этот метод сортирует список на месте и возвращает `None` (оболочка не показывает это значение).
❽ Теперь массив `fruits` отсортирован.

По умолчанию строки в Python сортируются лексикографически по кодам символов. Это означает, что заглавные буквы в кодировке ASCII предшествуют строчным, а порядок сортировки не-ASCII символов вряд ли будет сколько-нибудь разумным. В разделе «Сортировка текста в кодировке Unicode» главы 4 рассматривается вопрос о том, как сортировать текст в порядке, который представляется человеку естественным.

В отсортированной последовательности поиск производится очень эффективно. Стандартный алгоритм двоичного поиска уже имеется в модуле `bisect` из стандартной библиотеки Python. Этот модуль включает также вспомогатель-

[1] Основной алгоритм сортировки в Python называется в честь автора, Тима Петерса. Детали алгоритма Timsort обсуждаются во врезке «Поговорим» в конце этой главы.

[2] Слова в этом примере отсортированы по алфавиту, потому что содержат только строчные буквы в кодировке ASCII. См. предупреждение после примера.

ную функцию `bisect.insort`, которая гарантирует, что отсортированная последовательность такой и останется после вставки новых элементов. Иллюстрированное введение в модуль `bisect` имеется в статье «Managing Ordered Sequences with Bisect» (*https://www.fluentpython.com/extra/ordered-sequences-with-bisect/*) на сопроводительном сайте *fluentpython.com*.

Многое из описанного до сих пор относится к любым последовательностям, а не только к спискам или кортежам. Программисты на Python иногда чрезмерно увлекаются типом `list` просто потому, что он очень удобен, – знаю, сам грешен. Например, при работе со списками чисел лучше использовать массивы. Остаток этой главы посвящен альтернативам спискам и кортежам.

Когда список не подходит

Тип `list` гибкий и простой в использовании, но не всегда оптимален. Например, если требуется сохранить 10 миллионов чисел с плавающей точкой, то тип `array` будет гораздо эффективнее, поскольку в нем хранятся не полные объекты `float`, а только упакованные байты, представляющие их машинные значения, – как в массиве в языке C. С другой стороны, если вы часто добавляете и удаляете элементы из того или другого конца списка, стоит вспомнить о типе `deque` (двусторонняя очередь) – более эффективной структуре данных FIFO[1].

Если в программе много проверок на вхождение (например, `item in my_collection`), то, возможно, в качестве типа `my_collection` стоит взять `set`, особенно если количество элементов велико. Множества оптимизированы для быстрой проверки вхождения. Однако они не упорядочены и потому не являются последовательностями. Мы будем рассматривать множества в главе 3.

Массивы

Если список содержит только числа, то тип `array.array` эффективнее, чем `list`: он поддерживает все операции над изменяемыми последовательностями (включая `.pop`, `.insert` и `.extend`), а также дополнительные методы для быстрой загрузки и сохранения, например `.frombytes` и `.tofile`.

Массив Python занимает столько же памяти, сколько массив C. Как показано на рис. 2.1, в массиве значений типа `float` хранятся не полноценные экземпляры этого типа, а только упакованные байты, представляющие их машинные значения, – как в массиве `double` в языке C. При создании экземпляра `array` задается код типа – буква, определяющая, какой тип C использовать для хранения элементов.

Например, код типа `b` соответствует типу C `signed char`, описывающему целые числа от –128 до 127. Если создать массив `array('b')`, то каждый элемент будет храниться в одном байте и интерпретироваться как целое число. Если последовательность чисел велика, то это позволяет сэкономить много памяти. А Python не даст записать в массив число, не соответствующее заданному типу.

В примере 2.19 демонстрируется создание, сохранение и загрузка массива, содержащего 10 миллионов случайных чисел с плавающей точкой.

[1] First in, first out (первым пришел, первым ушел) – поведение очереди, подразумеваемое по умолчанию.

Пример 2.19. Создание, сохранение и загрузка большого массива чисел с плавающей точкой

```
>>> from array import array ❶
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7))) ❷
>>> floats[-1] ❸
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp) ❹
>>> fp.close()
>>> floats2 = array('d') ❺
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7) ❻
>>> fp.close()
>>> floats2[-1] ❼
0.07802343889111107
>>> floats2 == floats ❽
True
```

❶ Импортировать тип `array`.
❷ Создать массив чисел с плавающей точкой двойной точности (код типа `'d'`) из любого итерируемого объекта – в данном случае генераторного выражения.
❸ Прочитать последнее число в массиве.
❹ Сохранить массив в двоичном файле.
❺ Создать пустой массив чисел с плавающей точкой двойной точности.
❻ Прочитать 10 миллионов чисел из двоичного файла.
❼ Прочитать последнее число в массиве.
❽ Проверить, что содержимое обоих массивов совпадает.

Как видим, пользоваться методами `array.tofile` и `array.fromfile` легко. Выполнив этот пример, вы убедитесь, что и работают они очень быстро. Несложный эксперимент показывает, что для загрузки методом `array.fromfile` 10 миллионов чисел с плавающей точкой двойной точности из двоичного файла, созданного методом `array.tofile`, требуется примерно 0,1 с. Это почти в 60 раз быстрее чтения из текстового файла, когда требуется разбирать каждую строку встроенной функцией `float`. Метод `array.tofile` работает примерно в 7 раз быстрее, чем запись чисел с плавающей точкой в текстовый файл по одному на строку. Кроме того, размер двоичного файла с 10 миллионами чисел двойной точности составляет 80 000 000 байт (по 8 байт на число, с нулевыми накладными расходами), а текстового файла с теми же данными – 181 515 739 байт.

Для частных случаев числовых массивов, представляющих такие двоичные данные, как растровые изображения, в Python имеются типы `bytes` и `bytearray`, которые мы обсудим в главе 4.

Завершим этот раздел о массивах таблицей 2.3, в которой сравниваются свойства типов `list` и `array.array`.

Таблица 2.3. Методы и атрибуты типов list и array (нерекомендуемые методы массива, а также унаследованные от object, для краткости опущены)

	list	array	
`s.__add__(s2)`	●	●	`s + s2` – конкатенация
`s.__iadd__(s2)`	●	●	`s += s2` – конкатенация на месте
`s.append(e)`	●	●	Добавление элемента в конец списка
`s.byteswap()`		●	Перестановка всех байтов в массиве с целью изменения машинной архитектуры
`s.clear()`	●		Удаление всех элементов
`s.__contains__(e)`	●	●	`e` входит в `s`
`s.copy()`	●		Поверхностная копия списка
`s.__copy__()`		●	Поддержка метода `copy.copy`
`s.count(e)`	●	●	Подсчет числа вхождений элемента
`s.__deepcopy__()`		●	Оптимизированная поддержка метода `copy.deepcopy`
`s.__delitem__(p)`	●	●	Удаление элемента в позиции `p`
`s.extend(it)`	●	●	Добавление в конец списка элементов из итерируемого объекта `it`
`s.frombytes(b)`		●	Добавление в конец элементов из последовательности байтов, интерпретируемых как упакованные машинные слова
`s.fromfile(f, n)`		●	Добавление в конец `n` элементов из двоичного файла `f`, интерпретируемых как упакованные машинные слова
`s.fromlist(l)`		●	Добавление в конец элементов из списка; если хотя бы один возбуждает исключение `TypeError`, то не добавляется ничего
`s.__getitem__(p)`	●	●	`s[p]` – получение элемента в указанной позиции
`s.index(e)`	●	●	Поиск позиции первого вхождения `e`
`s.insert(p, e)`	●		Вставка элемента `e` перед элементом в позиции `p`
`s.itemsize`		●	Размер каждого элемента массива в байтах
`s.__iter__()`	●	●	Получение итератора
`s.__len__()`	●	●	`len(s)` – количество элементов
`s.__mul__(n)`	●	●	`s * n` – кратная конкатенация
`s.__imul__(n)`	●		`s *= n` – кратная конкатенация на месте
`s.__rmul__(n)`	●	●	`n * s` – инверсная кратная конкатенация[a]
`s.pop([p])`	●	●	Удалить и вернуть последний элемент или элемент в позиции `p`, если она задана
`s.remove(e)`	●	●	Удалить первое вхождение элемента `e`, заданного своим значением
`s.reverse()`	●	●	Изменить порядок элементов на противоположный на месте

Окончание табл. 2.3

	list	array	
`s.__reversed__()`	●	●	Получить итератор для перебора элементов от конца к началу
`s.__setitem__(p, e)`	●	●	`s[p] = e` – поместить `e` в позицию `p` вместо находящегося там элемента
`s.sort([key], [reverse])`	●		Отсортировать элементы на месте с факультативными аргументами `key` и `reverse`
`s.tobytes()`		●	Сохранение элементов как упакованных машинных слов в объекте типа `bytes`
`s.tofile(f)`		●	Сохранение элементов как упакованных машинных слов в двоичном файле `f`
`s.tolist()`		●	Сохранение элементов в виде числовых объектов в объекте `list`
`s.typecode`		●	Односимвольная строка, описывающая С-тип элементов

[a] Инверсные операторы рассматриваются в главе 16.

В версии Python 3.10 у типа `array` нет метода сортировки на месте, аналогичного `list.sort()`. Чтобы отсортировать массив, воспользуйтесь встроенной функцией `sorted`, которая перестраивает массив:

```
a = array.array(a.typecode, sorted(a))
```

Чтобы поддерживать массив в отсортированном состоянии при вставке элементов, пользуйтесь функцией `bisect.insort`.

Если вы часто работаете с массивами и ничего не знаете о типе `memoryview`, то много теряете в жизни. Читайте следующий раздел.

Представления областей памяти

Встроенный класс `memoryview` – это тип последовательности в общей памяти, который позволяет работать со срезами массивов, ничего не копируя. Он появился под влиянием библиотеки NumPy (которую мы обсудим ниже в разделе «NumPy»). Трэвис Олифант (Travis Oliphant), основной автор NumPy, на вопрос «Когда использовать memoryview?» (*https://stackoverflow.com/questions/4845418/when-should-a-memoryview-be-used/*) отвечает так:

> По существу, memoryview – это обобщенная структура массива NumPy, встроенная в сам язык Python (но без математических операций). Она позволяет разделять память между структурами данных (например, изображениями в библиотеке PIL, базами данных SQLlite, массивами NumPy и т. д.) без копирования. Для больших наборов данных это очень важно.

С применением нотации, аналогичной той, что используется в модуле `array`, метод `memoryview.cast` позволяет изменить способ чтения и записи нескольких байтов в виде блоков, не перемещая ни одного бита. Метод `memoryview.cast` возвращает другой объект `memoryview`, занимающий то же самое место в памяти.

В примере 2.20 показано, как создать различные представления одного и того же массива 6 байт, чтобы его можно было рассматривать как матрицу 2×3 или 3×2.

Пример 2.20. Обращение с 6 байтами в памяти как с представлениями матриц 1×6, 2×3 или 3×2

```
>>> from array import array
>>> octets = array('B', range(6)) ❶
>>> m1 = memoryview(octets) ❷
>>> m1.tolist()
[0, 1, 2, 3, 4, 5]
>>> m2 = m1.cast('B', [2, 3]) ❸
>>> m2.tolist()
[[0, 1, 2], [3, 4, 5]]
>>> m3 = m1.cast('B', [3, 2]) ❹
>>> m3.tolist()
[[0, 1], [2, 3], [4, 5]]
>>> m2[1,1] = 22 ❺
>>> m3[1,1] = 33 ❻
>>> octets ❼
array('B', [0, 1, 2, 33, 22, 51])
```

❶ Построить массив из шести байт (код типа `'B'`).
❷ Построить по этому массиву `memoryview`, а затем экспортировать его как список.
❸ Построить новое `memoryview` по предыдущему, но с 2 строками и 3 столбцами.
❹ Еще одно `memoryview`, на этот раз с 3 строками и 2 столбцами.
❺ Перезаписать байт в строке 1, столбце 1 представления `m2` значением 22.
❻ Перезаписать байт в строке 1, столбце 1 представления `m3` значением 33.
❼ Отобразить исходный массив, доказав тем самым, что `octets`, `m1`, `m2` и `m3` использовали одну и ту же память.

Впечатляющую мощь `memoryview` можно использовать и во вред.

В примере 2.21 показано, как изменить один байт в массиве 16-разрядных целых чисел.

Пример 2.21. Изменение значения элемента массива путем манипуляции одним из его байтов

```
>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers) ❶
>>> len(memv)
5
>>> memv[0] ❷
-2
>>> memv_oct = memv.cast('B') ❸
>>> memv_oct.tolist() ❹
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
>>> memv_oct[5] = 4 ❺
>>> numbers
array('h', [-2, -1, 1024, 1, 2]) ❻
```

❶ Построить объект `memoryview` по массиву пяти целых чисел типа `short signed` (код типа `'h'`).

❷ `memv` видит те же самые 5 элементов массива.
❸ Создать объект `memv_oct`, приведя элементы `memv` к коду типа 'B' (`unsigned char`).
❹ Экспортировать элементы `memv_oct` в виде списка для инспекции.
❺ Присвоить значение 4 байту со смещением 5.
❻ Обратите внимание, как изменились числа: двухбайтовое число, в котором старший байт равен 4, равно 1024.

Пример инспекции `memoryview` с помощью пакета `struct` можно найти на сайте *fluentpython.com* в статье «Parsing binary records with struct» (*https://www.fluentpython.com/extra/parsing-binary-struct/*).

А пока отметим, что для нетривиальных численных расчетов с применением массивов следует использовать библиотеки NumPy. Рассмотрим их прямо сейчас.

NumPy

В этой книге я стараюсь ограничиваться тем, что уже есть в стандартной библиотеке Python, и показывать, как извлечь из этого максимум пользы. Но библиотека NumPy – это такое чудо, что заслуживает небольшого отступления.

Именно чрезвычайно хорошо развитым операциям с массивами и матрицами в NumPy язык Python обязан признанием со стороны ученых, занимающихся вычислительными приложениями. В NumPy реализованы типы многомерных однородных массивов и матриц, в которых можно хранить не только числа, но и определенные пользователем записи. При этом предоставляются эффективные поэлементные операции.

Библиотека SciPy, написанная поверх NumPy, предлагает многочисленные вычислительные алгоритмы, относящиеся к линейной алгебре, численному анализу и математической статистике. SciPy работает быстро и надежно, потому что в ее основе лежит широко используемый код на C и Fortran из репозитория Netlib Repository (*http://www.netlib.org*). Иными словами, SciPy дает ученым лучшее из обоих миров: интерактивную оболочку и высокоуровневые API, присущие Python, и оптимизированные функции обработки числовой информации промышленного качества, написанные на C и Fortran.

В качестве очень простой демонстрации в примере 2.22 показаны некоторые операции с двумерными массивами в NumPy.

Пример 2.22. Простые операции со строками и столбцами из модуля numpy.ndarray

```
>>> import numpy as np ❶
>>> a = np.arange(12) ❷
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape ❸
(12,)
>>> a.shape = 3, 4 ❹
>>> a
```

```
array([[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11]])
>>> a[2] ❺
array([ 8, 9, 10, 11])
>>> a[2, 1] ❻
9
>>> a[:, 1] ❼
array([1, 5, 9])
>>> a.transpose() ❽
array([[ 0, 4, 8],
[ 1, 5, 9],
[ 2, 6, 10],
[ 3, 7, 11]])
```

❶ Импортировать NumPy, предварительно установив (этот пакет не входит в стандартную библиотеку Python).
❷ Построить и распечатать массив `numpy.ndarray`, содержащий целые числа от 0 до 11.
❸ Распечатать размерности массива: это одномерный массив с 12 элементами.
❹ Изменить форму массива, добавив еще одно измерение, затем распечатать результат.
❺ Получить строку с индексом 2.
❻ Получить элемент с индексами 2, 1.
❼ Получить столбец с индексом 1.
❽ Создать новый массив, транспонировав исходный (т. е. переставив местами строки и столбцы).

NumPy также поддерживает загрузку, сохранение и применение операций сразу ко всем элементам массива `numpy.ndarray`:

```
>>> import numpy
>>> floats = numpy.loadtxt('floats-10M-lines.txt') ❶
>>> floats[-3:] ❷
array([ 3016362.69195522, 535281.10514262, 4566560.44373946])
>>> floats *= .5 ❸
>>> floats[-3:]
array([ 1508181.34597761, 267640.55257131, 2283280.22186973])
>>> from time import perf_counter as pc ❹
>>> t0 = pc(); floats /= 3; pc() - t0 ❺
0.03690556302899495
>>> numpy.save('floats-10M', floats) ❻
>>> floats2 = numpy.load('floats-10M.npy', 'r+') ❼
>>> floats2 *= 6
>>> floats2[-3:] ❽
memmap([ 3016362.69195522, 535281.10514262, 4566560.44373946])
```

❶ Загрузить 10 миллионов чисел с плавающей точкой из текстового файла.
❷ С помощью нотации получения среза распечатать последние три числа.
❸ Умножить каждый элемент массива `floats` на 0.5 и снова распечатать последние три элемента.

❹ Импортировать таймер высокого разрешения (включен в стандартную библиотеку начиная с версии Python 3.3).
❺ Разделить каждый элемент на 3; для 10 миллионов чисел с плавающей точкой это заняло менее 40 миллисекунд.
❻ Сохранить массив в двоичном файле с расширением *.npy*.
❼ Загрузить данные в виде спроецированного на память файла в другой массив; это позволяет эффективно обрабатывать срезы массива, хотя он и не находится целиком в памяти.
❽ Умножить все элементы на 6 и распечатать последние три.

Этот код приведен, только чтобы разжечь ваш аппетит.

NumPy и SciPy – потрясающие библиотеки, лежащие в основе не менее замечательных библиотек для анализа данных, в т. ч. Pandas (*http://pandas.pydata.org*), которая предоставляет эффективные типы массивов для хранения нечисловых данных, а также функции импорта-экспорта, совместимые с различными форматами (например, CSV, XLS, дамп SQL, HDF5 и т. д.), и scikit-learn (*https://scikit-learn.org/stable/*), которая сейчас является самым широко распространенным набором инструментов для машинного обучения. Большинство функций в библиотеках NumPy и SciPy написаны на C или C++ и могут задействовать все доступные процессорные ядра, т. к. освобождают глобальную блокировку интерпретатора (GIL), присутствующую в Python. Проект Dask (*https://dask.org/*) также поддерживает распределение обработки с помощью NumPy, Pandas и scikit-learn между машинами, образующими кластер. Эти пакеты заслуживают отдельной книги, правда, не этой. Однако любой обзор последовательностей в Python был бы неполным без упоминания о массивах NumPy, хотя бы беглого.

Познакомившись с плоскими последовательностями – стандартными массивами и массивами NumPy, – обратимся к совершенно другой альтернативе старого доброго списка `list`: очередям.

Двусторонние и другие очереди

Методы `.append` и `.pop` позволяют использовать список `list` как стек или очередь (если вызывать только `.append` и `.pop(0)`, то получится дисциплина обслуживания LIFO). Однако вставка и удаление элемента из левого конца списка (с индексом 0) обходятся дорого, потому что приходится сдвигать весь список.

Класс `collections.deque` – это потокобезопасная двусторонняя очередь, предназначенная для быстрой вставки и удаления из любого конца. Эта структура удобна и для хранения списка «последних виденных элементов» и прочего в том же духе, т. к. `deque` можно сделать ограниченной (при создании задать максимальную длину). Тогда по заполнении `deque` добавление новых элементов приводит к удалению элементов с другого конца. В примере 2.23 показаны типичные операции со структурой `deque`.

Пример 2.23. Работа с очередью

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10) ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3) ❷
```

```
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1) ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33]) ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40]) ❺
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

❶ Необязательный аргумент `maxlen` задает максимальное число элементов в этом экземпляре `deque`, при этом устанавливается допускающий только чтение атрибут экземпляра `maxlen`.

❷ В результате циклического сдвига с `n > 0` элементы удаляются с правого конца и добавляются с левого; при `n < 0` удаление производится с левого конца, а добавление – с правого.

❸ При добавлении элемента в заполненную очередь (`len(d) == d.maxlen`) происходит удаление с другого конца; обратите внимание, что в следующей строке элемент 0 отсутствует.

❹ При добавлении трех элементов справа удаляются три элемента слева: –1, 1 и 2.

❺ Отметим, что функция `extendleft(iter)` добавляет последовательные элементы из объекта `iter` в левый конец очереди, т. е. в итоге элементы будут размещены в порядке, противоположном исходному.

В табл. 2.5 сравниваются методы классов `list` и `deque` (унаследованные от `object` не показаны).

Отметим, что `deque` реализует большинство методов `list` и добавляет несколько новых, связанных с ее назначением, например `popleft` и `rotate`. Но существует и скрытая неэффективность: удаление элементов из середины `deque` производится медленно. Эта структура данных оптимизирована для добавления и удаления элементов только с любого конца.

Операции `append` и `popleft` атомарны, поэтому `deque` можно безопасно использовать как FIFO-очередь в многопоточных приложениях без явных блокировок.

Таблица 2.4. Методы, реализованные в классах list и deque (унаследованные от object для краткости опущены)

	list	deque	
`s.__add__(s2)`	●		`s + s2` – конкатенация
`s.__iadd__(s2)`	●	●	`s += s2` – конкатенация на месте
`s.append(e)`	●	●	Добавление элемента справа (после последнего)
`s.appendleft(e)`		●	Добавление элемента слева (перед первым)

Окончание табл. 2.4

	list	deque	
`s.clear()`	●	●	Удаление всех элементов
`s.__contains__(e)`	●		`e` входит в `s`
`s.copy()`	●		Поверхностная копия списка
`s.__copy__()`		●	Поддержка `copy.copy` (поверхностная копия)
`s.count(e)`	●	●	Подсчет числа вхождений элемента
`s.__delitem__(p)`	●	●	Удаление элемента в позиции `p`
`s.extend(i)`	●	●	Добавление элементов из итерируемого объекта `it` справа
`s.extendleft(i)`		●	Добавление элементов из итерируемого объекта `it` слева
`s.__getitem__(p)`	●	●	`s[p]` – получение элемента в указанной позиции
`s.index(e)`	●		Поиск позиции первого вхождения `e`
`s.insert(p, e)`	●		Вставка элемента `e` перед элементом в позиции `p`
`s.__iter__()`	●	●	Получение итератора
`s.__len__()`	●	●	`len(s)` – количество элементов
`s.__mul__(n)`	●		`s * n` – кратная конкатенация
`s.__imul__(n)`	●		`s *= n` – кратная конкатенация на месте
`s.__rmul__(n)`	●	●	`n * s` – инверсная кратная конкатенация[a]
`s.pop()`	●	●	Удалить и вернуть последний элемент[b]
`s.popleft()`		●	Удалить и вернуть первый элемент
`s.remove(e)`	●	●	Удалить первое вхождение элемента `e`, заданного своим значением
`s.reverse()`	●	●	Изменить порядок элементов на противоположный на месте
`s.__reversed__()`	●	●	Получить итератор для перебора элементов от конца к началу
`s.rotate(n)`		●	Переместить `n` элементов из одного конца в другой
`s.__setitem__(p, e)`	●	●	`s[p] = e` – поместить `e` в позицию `p` вместо находящегося там элемента
`s.sort([key], [reverse])`	●		Отсортировать элементы на месте с факультативными аргументами `key` и `reverse`

[a] Инверсные операторы рассматриваются в главе 16.

[b] Вызов `a_list.pop(p)` позволяет удалить элемент в позиции `p`, но класс `deque` его не поддерживает.

Помимо `deque`, в стандартной библиотеке Python есть пакеты, реализующие другие виды очередей.

`queue`

Содержит синхронизированные (т. е. потокобезопасные) классы `Queue`, `LifoQueue` и `PriorityQueue`. Они используются для безопасной коммуникации между потоками. Все три очереди можно сделать ограниченными, передав конструктору аргумент `maxsize`, больший 0. Однако, в отличие от `deque`, в случае переполнения элементы не удаляются из очереди, чтобы освободить место, а блокируется вставка новых элементов, т. е. программа ждет, пока какой-нибудь другой поток удалит элемент из очереди. Это полезно для ограничения общего числа работающих потоков.

`multiprocessing`

Реализует собственную неограниченную очередь `SimpleQueue` и ограниченную очередь `Queue`, очень похожие на аналоги в пакете `queue`, но предназначенные для межпроцессного взаимодействия. Чтобы упростить управление задачами, имеется также специализированный класс `multiprocessing.JoinableQueue`.

`asyncio`

Предоставляет классы `Queue`, `LifoQueue`, `PriorityQueue` и `JoinableQueue`, API которых построен по образцу классов из модулей `queue` и `multiprocessing`, но адаптирован для управления задачами в асинхронных программах.

`heapq`

В отличие от трех предыдущих модулей, `heapq` не содержит класса очереди, а предоставляет функции, в частности `heappush` и `heappop`, которые дают возможность работать с изменяемой последовательностью как с очередью с приоритетами, реализованной в виде пирамиды.

На этом мы завершаем обзор альтернатив типу `list` и изучение типов последовательностей в целом – за исключением особенностей типа `str` и двоичных последовательностей, которым посвящена отдельная глава 4.

Резюме

Свободное владение типами последовательностей из стандартной библиотеки – обязательное условие написания краткого, эффективного и идиоматичного кода на Python.

Последовательности Python часто классифицируются как изменяемые или неизменяемые, но полезно иметь в виду и другую классификацию: плоские и контейнерные последовательности. Первые более компактные, быстрые и простые в использовании, но в них можно хранить только атомарные данные, т. е. числа, символы и байты. Контейнерные последовательности обладают большей гибкостью, но могут стать источником сюрпризов при хранении в них изменяемых объектов, поэтому при использовании их для размещения иерархических структур данных следует проявлять осторожность.

К сожалению, в Python нет по-настоящему неизменяемой последовательности контейнерного типа: даже в «неизменяемых» кортежах значения могут быть изменены, если представляют собой изменяемые объекты, например списки или объекты, определенные пользователем.

Списковые включения и генераторные выражения – эффективный способ создания и инициализации последовательностей. Если вы еще не освоили эти

конструкции, потратьте какое-то время на изучение базовых способов их применения. Это нетрудно и очень скоро воздастся сторицей.

У кортежей в Python двоякая роль: записи с неименованными полями и неизменяемые списки. Используя кортеж в качестве неизменяемого списка, помните, что значение кортежа будет фиксировано, только если все его элементы также неизменяемы. Вызов функции `hash(t)` для кортежа – простой способ убедиться в том, что его значение никогда не изменится. Если `t` содержит изменяемые элементы, то будет возбуждено исключение `TypeError`.

Когда кортеж используется как запись, операция его распаковки – самый безопасный и понятный способ получить отдельные поля. Конструкция `*` работает не только с кортежами, но также со списками и итерируемыми объектами во многих контекстах. Некоторые способы ее применения появились в Python 3.5 и описаны в документе PEP 448 «Additional Unpacking Generalizations» (*https://peps.python.org/pep-0448/*). В Python 3.10 добавлен механизм сопоставления с образцом `match/case`, поддерживающий более развитые средства распаковки, называемые деструктуризацией.

Получение среза последовательности – одна из самых замечательных синтаксических конструкций Python, причем многие даже не знают всех ее возможностей. Многомерные срезы и нотация многоточия (...), нашедшие применение в NumPy, могут поддерживаться и другими пользовательскими последовательностями. Присваивание срезу – очень выразительный способ модификации изменяемых последовательностей.

Кратная конкатенация (`seq * n`) – удобный механизм и при должной осторожности может применяться для инициализации списка списков, содержащих изменяемые элементы. Операции составного присваивания `+=` и `*=` ведут себя по-разному для изменяемых и неизменяемых последовательностей. В последнем случае они по необходимости создают новую последовательность. Но если конечная последовательность изменяемая, то обычно она модифицируется на месте, хотя и не всегда, т. к. это зависит от того, как последовательность реализована.

Метод `sort` и встроенная функция `sorted` просты в использовании и обладают большой гибкостью благодаря необязательному аргументу `key`, который представляет собой функцию для вычисления критерия сортировки. Кстати, в качестве `key` могут выступать и встроенные функции `min` и `max`.

Помимо списков и кортежей, в стандартной библиотеке Python имеется класс `array.array`. И хотя пакеты NumPy и SciPy не входят в стандартную библиотеку, настоятельно рекомендуется хотя бы бегло познакомиться с ними любому, кто занимается численным анализом больших наборов данных.

В конце главы мы рассмотрели практичный потокобезопасный класс `collections.deque`, сравнили его API с API класса `list` (табл. 2.4) и кратко упомянули другие реализации очереди, имеющиеся в стандартной библиотеке.

Дополнительная литература

В главе 1 «Структуры данных» книги David Beazley, Brian K. Jones «*Python Cookbook*», 3-е издание (O'Reilly), имеется много рецептов, посвященных по-

следовательностям, в том числе рецепт 1.11 «Именованные срезы», из которого я позаимствовал присваивание срезов переменным для повышения удобочитаемости кода (пример 2.13).

Второе издание книги «*Python Cookbook*» охватывает версию Python 2.4, но значительная часть приведенного в ней кода работает и в Python 3, а многие рецепты в главах 5 и 6 относятся к последовательностям. Книгу редактировали Алекс Мартелли, Анна Мартелли Равенскрофт и Дэвид Эшер, свой вклад в нее внесли также десятки других питонистов. Третье издание было переписано с нуля и в большей степени ориентировано на семантику языка – особенно на изменения, появившиеся в Python 3, – тогда как предыдущие издания посвящены в основном прагматике (т. е. способам применения языка для решения практических задач). И хотя кое-какой код из второго издания уже нельзя считать наилучшим подходом, я все же полагаю, что полезно иметь под рукой оба издания «*Python Cookbook*».

В официальном документе о сортировке в Python «Sorting HOW TO» (*http://docs.python.org/3/howto/sorting.html*) приведено несколько примеров продвинутого применения `sorted` и `list.sort`.

Документ PEP 3132 «Extended Iterable Unpacking» (*http://python.org/dev/peps/pep-3132/*) – канонический источник сведений об использовании новой конструкции `*extra` в правой части параллельного присваивания. Если вам интересна история развития Python, то загляните в обсуждение проблемы «Missing *-unpacking generalizations» (*http://bugs.python.org/issue2292*), где предлагается еще более общее использование нотации распаковки итерируемых объектов. Документ PEP 448 «Additional Unpacking Generalizations» (*https://www.python.org/dev/peps/pep-0448/*) появился в результате этого обсуждения.

Как я говорил в разделе «Сравнение с последовательностью-образцом», написанный Кэролом Уиллингом раздел «Структурное сопоставление с образцом» (*https://docs.python.org/3.10/whatsnew/3.10.html*) главы «Что нового в Python 3.10» официальной документации – прекрасное введение в этот новый механизм, занимающее где-то 1400 слов (меньше 5 страниц в PDF-файле, который Firefox строит по HTML-коду). Документ PEP 636 «Structural Pattern Matching: Tutorial» (*https://peps.python.org/pep-0636/*) тоже неплох, но длиннее. Однако в нем имеется приложение A «Краткое введение» (*https://peps.python.org/pep-0636/#appendix-a-quick-intro*). Оно короче введения Уиллинга, потому что не содержит общих рассуждений на тему, почему сопоставление с образцом – это хорошо. Если вам нужны еще аргументы, чтобы убедить себя или других в пользе сопоставления с образцом, почитайте 22-страничный документ PEP 635 «Structural Pattern Matching: Motivation and Rationale» (*https://peps.python.org/pep-0635/*).

Статья в блоге Эли Бендерского «Less Copies in Python with the Buffer Protocol and memoryviews» (*https://eli.thegreenplace.net/2011/11/28/less-copies-in-python-with-the-buffer-protocol-and-memoryviews/*) содержит краткое пособие по использованию `memoryview`.

На рынке есть немало книг, посвященных NumPy, и в названиях некоторых из них слово «NumPy» отсутствует. Одна из них – книга Jake VanderPlas «Python Data Science Handbook» (*https://jakevdp.github.io/PythonDataScienceHandbook/*), вы-

ложенная в открытый доступ, другая – Wes McKinney «*Python for Data Analysis*» (*https://www.oreilly.com/library/view/python-for-data/9781491957653/*)[1].

«NumPy – это о векторизации». Так начинается находящаяся в открытом доступе книга Nicolas P. Rougier «From Python to NumPy» (*https://www.labri.fr/perso/nrougier/from-python-to-numpy/*). Векторизованные операции применяют математические функции ко всем элементам массива без необходимости писать явный цикл на Python. Они могут работать параллельно, пользуясь специальными командами современных процессоров, и либо задействовать несколько ядер, либо делегировать работу графическому процессору – в зависимости от библиотеки. Первый же пример в книге Руже демонстрирует ускорение в 500 раз после рефакторинга идиоматического класса на Python, пользующегося генератором, в компактную и эффективную функцию, вызывающую две векторные функции из библиотеки NumPy.

Если хотите научиться использовать класс `deque` (и другие коллекции), познакомьтесь с примерами и практическими рецептами в главе «Контейнерные типы данных» (*https://docs.python.org/3/library/collections.html*) документации по Python.

Лучшие аргументы в поддержку исключения последнего элемента диапазона и среза привел сам Эдсгер В. Дейкстра в короткой заметке под названием «Why Numbering Should Start at Zero» (*https://www.cs.utexas.edu/users/EWD/ transcriptions/EWD08xx/EWD831.html*). Тема этой заметки – математическая нотация, но она относится и к Python, потому что профессор Дейкстра строго и с юмором объясняет, почему последовательность 2, 3, …, 12 следует описывать только условием $2 \leqslant i < 13$. Все прочие разумные соглашения опровергаются, как и мысль о том, чтобы позволить пользователю самому выбирать соглашение. Название заметки наводит на мысль об индексировании с нуля, но на самом деле речь в ней идет о том, почему `'ABCDE'[1:3]` должно означать `'BC'`, а не `'BCD'`, и почему диапазон 2, 3, …, 12 следует записывать в виде `range(2, 13)`. Кстати, заметка рукописная, но вполне разборчивая. Почерк Дейкстры настолько четкий, что кто-то даже создал на его основе шрифт (*https://www.fonts101.com/fonts/view/Uncategorized/34398/Dijkstra*).

Поговорим

О природе кортежей

В 2012 году я презентовал плакат, касающийся языка ABC на конференции PyCon US. До создания Python Гвидо работал над интерпретатором языка ABC, поэтому пришел посмотреть на мой плакат. По ходу дела мы поговорили о составных объектах в ABC, которые, безусловно, являются предшественниками кортежей Python. Составные объекты также поддерживают параллельное присваивание и используются в качестве составных ключей словарей (в ABC они называются таблицами). Однако составные объекты не являются последовательностями. Они не допускают итерирования, к отдельному полю объекта нельзя обратиться по индексу, а уж тем более получить срез. Составной объект можно либо обрабатывать целиком, либо выделить поля с помощью параллельного присваивания – вот и всё.

[1] *Уэс Маккини.* Python и анализ данных. 2-е изд. ДМК Пресс, 2020 // *https://dmkpress.com/catalog/computer/programming/python/978-5-97060-590-5/*

Я сказал Гвидо, что в силу этих ограничений основная цель составных объектов совершенно ясна: это просто записи с неименованными полями. И вот что он ответил: «То, что кортежи ведут себя как последовательности, – просто хак».

Это иллюстрация прагматического подхода, благодаря которому Python оказался настолько удачнее и успешнее ABC. С точки зрения разработчика языка, заставить кортежи вести себя как последовательности почти ничего не стоит. В результате основное использование кортежей как записей не столь очевидно, но мы получили неизменяемые списки – пусть даже имя типа не такое говорящее, как `frozenlist`.

Плоские и контейнерные последовательности

Чтобы подчеркнуть различие между моделями памяти в последовательностях разных типов, я воспользовался терминами *контейнерная* и *плоская последовательность*. Слово «контейнер» употребляется в документации по модели данных (*https://docs.python.org/3/reference/datamodel.html#objects-values-and-types*):

> Некоторые объекты содержат ссылки на другие объекты, они называются контейнерами.

Я остановился на термине «контейнерная последовательность» для большей точности, потому что в Python есть контейнеры, не являющиеся последовательностями, например `dict` и `set`. Контейнерные последовательности могут быть вложенными, поскольку могут содержать объекты любого типа, в том числе своего собственного.

С другой стороны, *плоские последовательности* не могут быть вложенными, потому что в них разрешено хранить только простые атомарные типы, например целые, числа с плавающей точкой или символы.

Я выбрал термин *плоская последовательность*, потому что нуждался в чем-то, противоположном «контейнерной последовательности».

Несмотря на цитированное выше употребление слова «контейнеры» в официальной документации, в модуле `collections.abc` имеется класс с именем `Container`. В этом ABC есть всего один метод, `__contains__`, – специальный метод, поддерживающий оператор `in`. Это означает, что строки и массивы, не являющиеся контейнерами в традиционном смысле, являются тем не менее виртуальными подклассами `Container`, потому что реализуют метод `__contains__`. Это лишний раз подтверждает, что люди часто используют одно и то же слово в разных смыслах. В этой книге я пишу «контейнер» (container) строчными буквами, имея в виду «объект, содержащий ссылки на другие объекты», и `Container` с заглавной буквы и моноширинным шрифтом, когда хочу сослаться на класс `collections.abc.Container`.

Смешанные списки

В учебниках Python для начинающих подчеркивается, что списки могут содержать объекты разных типов, но на практике такая возможность не слишком полезна: ведь мы помещаем элементы в список, чтобы впоследствии их обработать, а это значит, что все элементы должны поддерживать общий набор операций (т. е. должны «крякать», даже если не родились утками). Например, в Python 3 невозможно отсортировать список, если его элементы не сравнимы:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

В отличие от списков, кортежи часто содержат элементы разных типов. И это естественно: если каждый элемент кортежа – поле, то поля могут иметь различные типы.

Аргумент key – истинный бриллиант

Факультативный аргумент `key` метода `list.sort` и функций `sorted`, `max` и `min` – отличная идея. В других языках вы должны передавать функцию сравнения с двумя аргументами, как, например, ныне не рекомендуемая функция `cmp(a, b)` в Python 2. Но использовать `key` и проще, и эффективнее. Проще – потому что нужно определить функцию всего с одним аргументом, которая извлекает или вычисляет критерий, с помощью которого сортируются объекты; это легче, чем написать функцию с двумя аргументами, возвращающую –1, 0 или 1. А эффективнее – потому что функция `key` вызывается только один раз для каждого элемента, тогда как функция сравнения с двумя аргументами – всякий раз, как алгоритму сортировки необходимо сравнить два элемента. Разумеется, Python тоже должен сравнивать ключи во время сортировки, но это сравнение производится в оптимизированном коде на C, а не в написанной вами функции Python.

Кстати, аргумент `key` даже позволяет сортировать списки, содержащие числа и похожие на числа строки. Нужно только решить, как интерпретировать все объекты: как целые числа или как строки:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l, key=int)
[0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
>>> sorted(l, key=str)
[0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
```

Oracle, Google и таинственный Timbot

В функции `sorted` и методе `list.sort` используется адаптивный алгоритм Timsort, который переключается с сортировки вставками на сортировку слиянием в зависимости от того, как упорядочены данные. Это эффективно, потому что в реальных данных часто встречаются уже отсортированные участки. На эту тему есть статья в Википедии (*http://en.wikipedia.org/wiki/Timsort*).

Алгоритм Timsort впервые был реализован в CPython в 2002 году. Начиная с 2009 года Timsort используется также для сортировки массивов в стандартном компиляторе Java и в Android, этот факт стал широко известен, потому что корпорация Oracle использовала относящийся к Timsort код как доказательство нарушения Google прав интеллектуальной собственности компании Sun. См., например, постановление судьи Уильяма Олсапа, вынесенное в 2012 году (*http://www.groklaw.net/pdf3/OraGoogle-1202.pdf*). В 2021 году Верховный суд США постановил, что использование компанией Google кода Java следует считать правомерным.

Алгоритм Timsort изобрел Тим Питерс, один из разработчиков ядра Python, настолько плодовитый, что его считали даже искусственным интеллектом – Timbot. Об этой конспирологической теории можно прочитать на страничке Python Humor (*https://www.python.org/doc/humor/#id9*). Тим – также автор «Дзен Python»: `import this`.

Глава 3

Словари и множества

> Python – по сути своей словари, обернутые тоннами синтаксического сахара.
>
> – Лало Мартинс, один из первых цифровых номадов и питонистов

Мы используем словари во всех своих программах на Python. Если не напрямую в коде, то косвенно, потому что тип `dict` – фундаментальная часть реализации Python. Атрибуты классов и экземпляров, пространства имен модулей, именованные аргументы функции – вот лишь некоторые фундаментальные конструкции, в которых используются словари. Встроенные типы, объекты и функции хранятся в словаре `__builtins__.__dict__`.

В силу своей важности словари в Python высокооптимизированы. В основе высокопроизводительных словарей лежат *хеш-таблицы*.

Хеш-таблицы лежат и в основе других встроенных типов: `set` и `frozenset`. Они предлагают более развитые API и наборы операторов, чем множества, встречающиеся в других популярных языках. В частности, множества в Python реализуют все основные теоретико-множественные операции: объединение, пересечение, проверку на подмножество и т. д. С их помощью мы можем выражать алгоритмы более декларативным способом, избегая вложенных циклов и условий.

Вот краткое содержание этой главы:

- современные синтаксические конструкции для построения и работы со словарями `dict` и отображениями, включая улучшенную распаковку и сопоставление с образцом;
- часто используемые методы типов отображений;
- специальная обработка отсутствия ключа;
- различные вариации типа `dict` в стандартной библиотеке;
- типы `set` и `frozenset`;
- как устройство хеш-таблиц отражается на поведении множеств и словарей.

Что нового в этой главе

Большинство изменений во втором издании связаны с новыми возможностями, относящимися к типам отображений.

- В разделе «Современный синтаксис словарей» рассматривается улучшенный синтаксис распаковки и различные способы объединения отображения, включая операторы `|` и `|=`, поддерживаемые классом `dict`, начиная с версии Python 3.9.

- В разделе «Сопоставление с отображениями-образцами» иллюстрируется использование отображений совместно с `match/case`, появившееся в версии Python 3.10.
- Раздел «collections.OrderedDict» теперь посвящен небольшим, но все еще сохраняющимся различиям между `dict` и `OrderedDict` – с учетом того, что начиная с версии Python 3.6 ключи в `dict` хранятся в порядке вставки.
- Добавлены новые разделы об объектах представлений, возвращаемых атрибутами `dict.keys`, `dict.items` и `dict.values`: «Представления словарей» и «Теоретико-множественные операции над представлениями словарей».

В основе реализации `dict` и `set` по-прежнему лежат хеш-таблицы, но в код `dict` внесено две важные оптимизации, позволяющие сэкономить память и сохранить порядок вставки ключей. В разделах «Практические последствия внутреннего устройства класса dict» и «Практические последствия внутреннего устройства класса set» сведено то, что нужно знать для их эффективного использования.

Добавив более 200 страниц во второе издание, я переместил раздел «Внутреннее устройство множеств и словарей» на сопроводительный сайт *fluentpython.com*. Дополненная и исправленная 18-страничная статья включает пояснения и диаграммы, касающиеся следующих вопросов:

- алгоритм и структуры данных хеш-таблиц, начиная с более простого для понимания использования в классе `set`;
- оптимизация памяти, обеспечивающая сохранение порядка вставки ключей в экземпляры `dict` (начиная с Python 3.6);
- обеспечивающее разделение ключей размещение в памяти словарей, в которых хранятся атрибуты экземпляра – атрибут `__dict__` определенных пользователем объектов (оптимизация, реализованная в Python 3.3).

Современный синтаксис словарей

В следующих разделах описываются средства построения, распаковки и обработки отображений. Некоторые из них присутствуют в языке давно, но, возможно, будут новыми для вас. Для других необходима версия Python 3.9 (например, для оператора `|`) или 3.10 (для `match/case`). Но начнем с описания самых старых и самых лучших средств.

Словарные включения

Начиная с версии Python 2.7 синтаксис списковых выключений и генераторных выражений расширен на словарные включения (а также на множественные включения, о которых речь ниже). *Словарное включение* (dictcomp) строит объект `dict`, порождая пары `key:value` из произвольного итерируемого объекта. В примере 3.1 демонстрируется применение словарного включения для построения двух словарей из одного и того же списка кортежей.

Пример 3.1. Примеры словарных включений

```
>>> dial_codes = [ ❶
...     (880, 'Bangladesh'),
...     (55, 'Brazil'),
...     (86, 'China'),
...     (91, 'India'),
...     (62, 'Indonesia'),
...     (81, 'Japan'),
...     (234, 'Nigeria'),
...     (92, 'Pakistan'),
...     (7, 'Russia'),
...     (1, 'United States'),
... ]
>>> country_dial = {country: code for code, country in dial_codes} ❷
>>> country_dial
{'Bangladesh': 880, 'Brazil': 55, 'China': 86, 'India': 91, 'Indonesia': 62,
'Japan': 81, 'Nigeria': 234, 'Pakistan': 92, 'Russia': 7, 'United States': 1}
>>> {code: country.upper() ❸
...     for country, code in sorted(country_dial.items())
...     if code < 70}
{55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA', 1: UNITED STATES'}
```

❶ Итерируемый объект `dial_codes`, содержащий список пар ключ-значение, можно напрямую передать конструктору `dict`, но...

❷ ... мы инвертируем пары: ключом является `country`, а значением – `code`.

❸ Сортируем `country_dial` по названию страны, снова инвертируем пары, преобразуем значения в верхний регистр и оставляем только элементы, для которых `code < 70`.

Если вы уже освоили списковые включения, то словарные естественно станут следующим шагом. Если нет, то тем больше причин поскорее заняться этим – ведь синтаксис списковых включений теперь обобщен.

Распаковка отображений

Документ PEP 448 «Additional Unpacking Generalizations» (*https://peps.python.org/pep-0448/*) ввел два дополнения в поддержку распаковки отображений сверх того, что было в Python 3.5.

Во-первых, оператор `**` можно применять более чем к одному аргументу вызванной функции. Это допустимо, когда все ключи являются строками и среди аргументов нет повторяющихся ключей (т. к. дубликаты именованных аргументов запрещены).

```
>>> def dump(**kwargs):
...     return kwargs
...
>>> dump(**{'x': 1}, y=2, **{'z': 3})
{'x': 1, 'y': 2, 'z': 3}
```

Во-вторых, оператор `**` можно использовать внутри словарного литерала – и тоже несколько раз:

```
>>> {'a': 0, **{'x': 1}, 'y': 2, **{'z': 3, 'x': 4}}
{'a': 0, 'x': 4, 'y': 2, 'z': 3}
```