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.

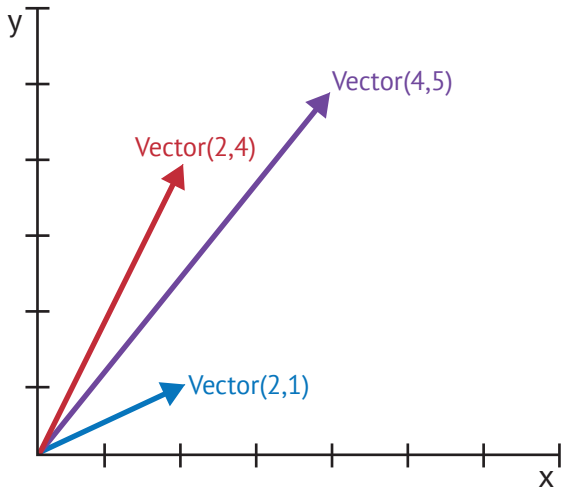


Рис. 1.1. Пример сложения двумерных векторов: Vector(2, 4) + Vector(2, 1) = Vector(4, 5)

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.

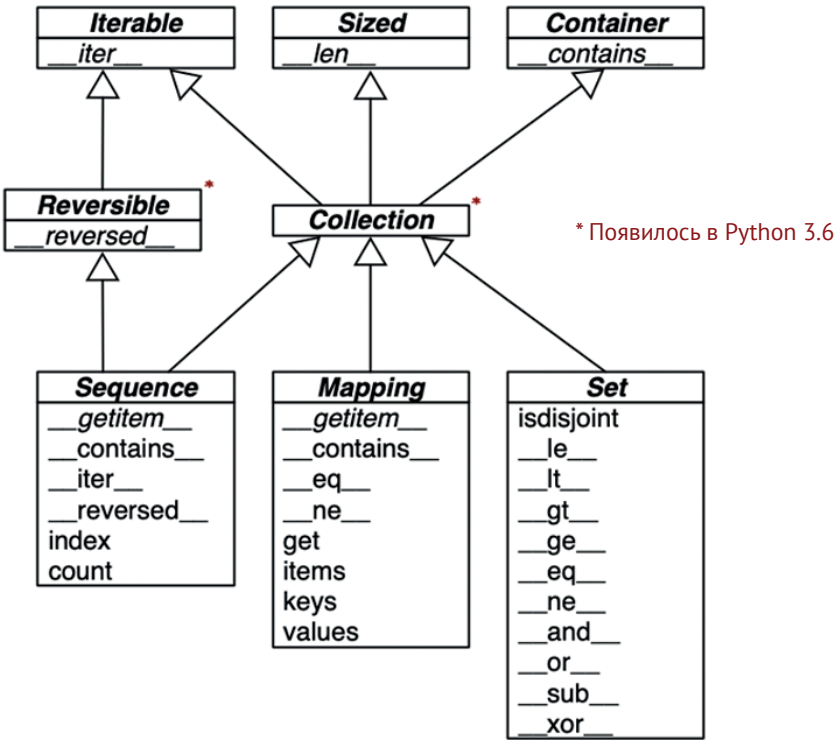


Рис. 1.2. UML-диаграмма классов, иллюстрирующая наиболее важные типы коллекций. Методы, имена которых набраны курсивом, абстрактные, поэтому должны быть реализованы в конкретных подклассах, например `list` и `dict`. У остальных методов имеются конкретные реализации, которые подклассы могут унаследовать

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.

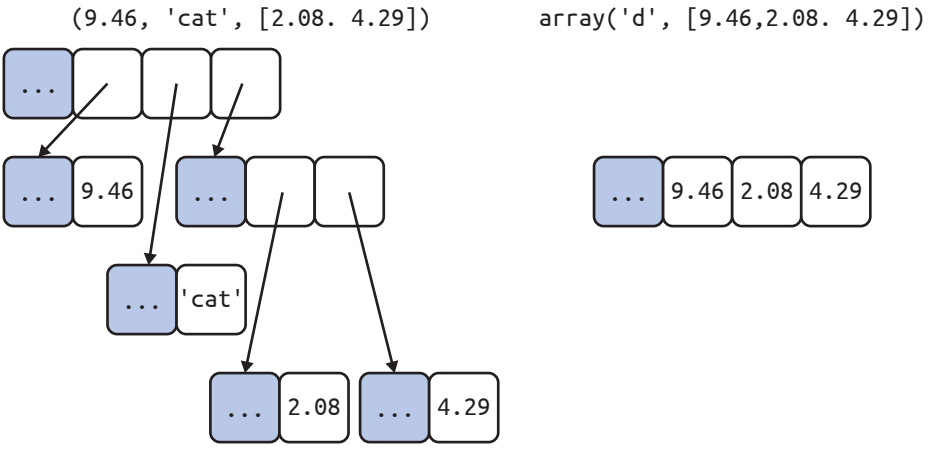


Рис. 2.1. Упрощенные диаграммы размещения кортежа и массива в памяти (тот и другой содержат три элемента). Закрашенные ячейки представляют заголовок объекта Python в памяти – пропорции не соблюдены. В кортеже хранится массив ссылок на элементы. Каждый элемент – отдельный объект Python, быть может, содержащий 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
```

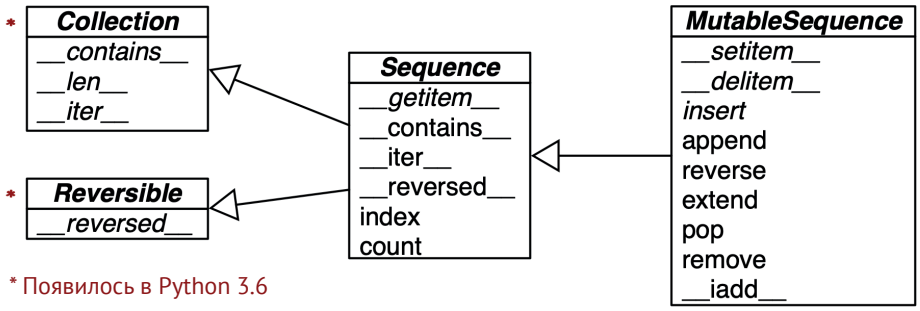


Рис. 2.2. Упрощенная UML-диаграмма нескольких классов из модуля `collections.abc` суперклассы показаны слева, стрелки ведут от подклассов к суперклассам, курсивом набраны имена абстрактных классов и абстрактных методов)

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.

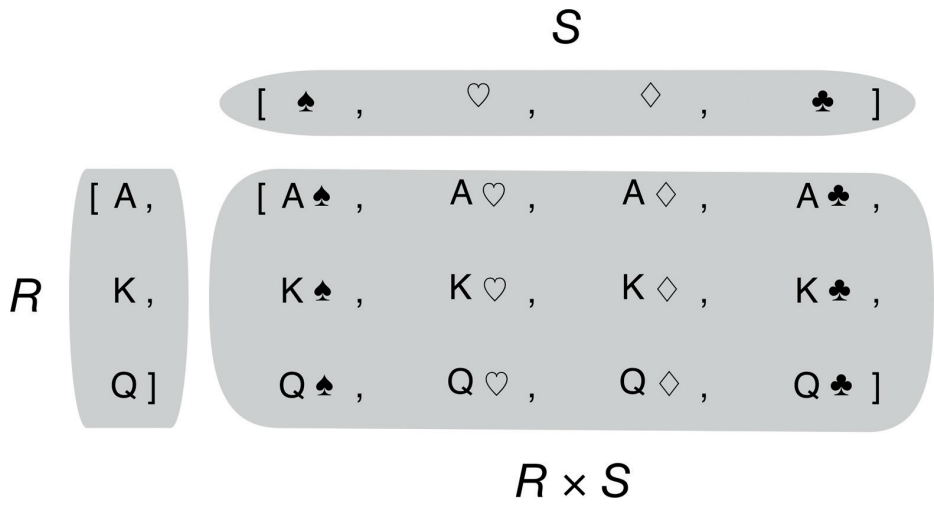


Рис. 2.3. Декартово произведение последовательности трех достоинств карт и последовательности четырех мастей дает последовательность, состоящую из двенадцати пар

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.

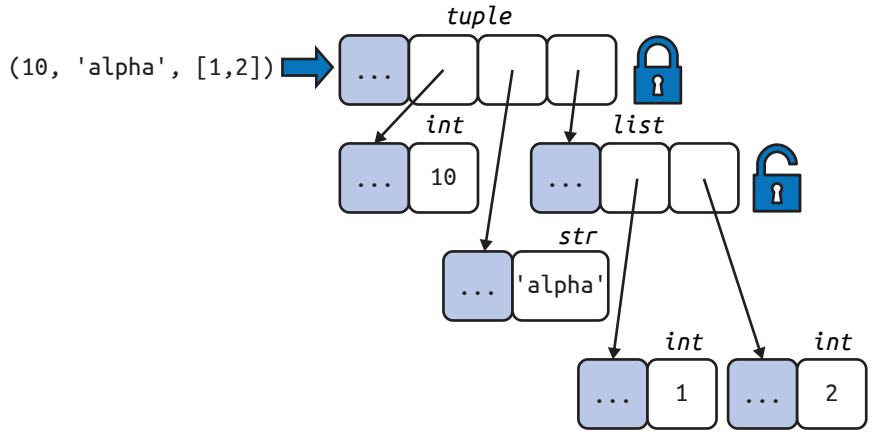


Рис. 2.4. Само содержимое кортежа неизменно, но это лишь означает, что хранящиеся в кортеже ссылки всегда указывают на одни и те же объекты. Однако если какой-то из этих объектов изменяемый, например является списком, то его содержимое может измениться

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 `+=`.

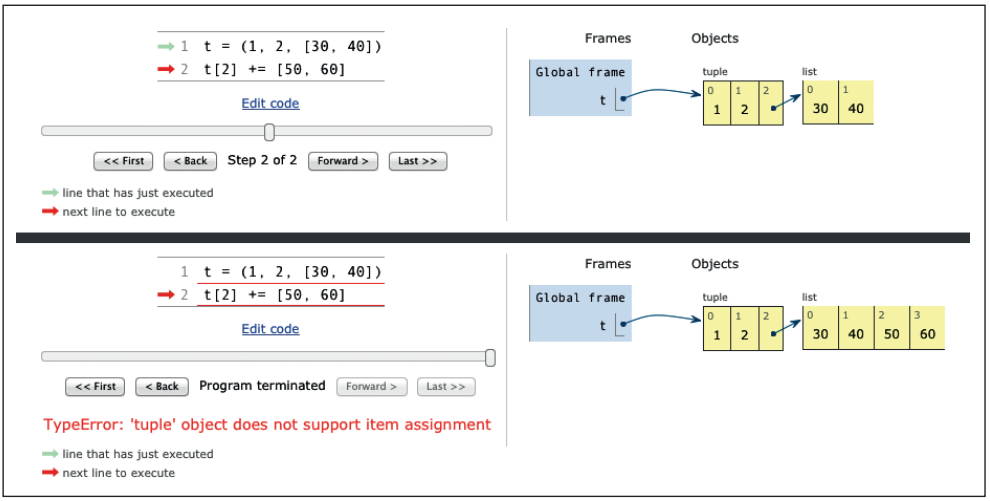


Рис. 2.5. Начальное и конечное состояния кортежа в задаче о присваивании (диаграммы сгенерированы на сайте Online Python Tutor)

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}
```

В этом случае повторяющиеся ключи разрешены. Последующее вхождение ключа перезаписывает предыдущее (как в случае значения `x` в примере выше).

Такой синтаксис можно использовать и для объединения отображений, но есть и другие способы. Читайте дальше.

Объединение отображений оператором |

В Python 3.9 поддерживаются операторы `|` и `|=` для объединения отображений. Это и понятно, потому что они же используются и для объединения множеств.

Оператор `|` создает новое отображение:

```
>>> d1 = {'a': 1, 'b': 3}
>>> d2 = {'a': 2, 'b': 4, 'c': 6}
>>> d1 | d2
{'a': 2, 'b': 4, 'c': 6}
```

Обычно тип нового отображения совпадает с типом левого операнда, `d1` в примере выше, но может совпадать и с типом правого, если в операции участвуют определенные пользователем типы. При этом действуют правила перегрузки операторов, которые мы будем рассматривать в главе 16.

Для модификации уже имеющегося отображения на месте служит оператор `|=`. Продолжим предыдущий пример – содержимое `d1` изменилось, хотя переменная осталась той же:

```
>>> d1
{'a': 1, 'b': 3}
>>> d1 |= d2
>>> d1
{'a': 2, 'b': 4, 'c': 6}
```

Если необходимо поддерживать работоспособность кода в версии Python 3.8 или более ранней, то в разделе «Motivation» документа PEP 584 «Add Union Operators To dict» (*https://peps.python.org/pep-0584/#motivation*) вы найдете хороший обзор других способов объединения отображений.

Теперь посмотрим, как к отображениям применяется сопоставление с образцом.

Сопоставление с отображением-образцом

Предложение `match/case` поддерживает субъекты, являющиеся отображениями. Отображения-образцы выглядят как литералы типа `dict`, но могут сопоставляться с экземплярами любого реального или виртуального подкласса `collections.abc.Mapping`[1].

[1] Виртуальным называется подкласс, зарегистрированный методом `.register()` абстрактного базового класса (см. раздел «Виртуальный подкласс ABC» главы 13). Типы, реализованные с помощью Python/C API, также допустимы, если установлен специальный бит признака. См. Py_TPFLAGS_MAPPING (*https://docs.python.org/3.10/c-api/typeobj.html#Py_TPFLAGS_MAPPING*).

В главе 2 мы занимались только последовательностями-образцами, но разные типы образцов можно комбинировать и вкладывать. Благодаря деструктуризации сопоставление с образцом является мощным средством обработки записей, состоящих из вложенных отображений и последовательностей. Это часто бывает полезно при чтении из JSON API и баз данных с полуструктурированными схемами, например MongoDB, EdgeDB или PostgreSQL. См. пример 3.2. Из простых аннотаций типов в функции `get_creators` следует, что функция принимает `dict` и возвращает `list`.

Пример 3.2. creator.py: `get_creators()` выделяет имена авторов из записей о произведениях

```
def get_creators(record: dict) -> list:
    match record:
        case {'type': 'book', 'api': 2, 'authors': [*names]}: ❶
            return names
        case {'type': 'book', 'api': 1, 'author': name}: ❷
            return [name]
        case {'type': 'book'}: ❸
            raise ValueError(f"Invalid 'book' record: {record!r}")
        case {'type': 'movie', 'director': name}: ❹
            return [name]
        case _: ❺
            raise ValueError(f'Invalid record: {record! r}')
```

❶ Сопоставить с любым отображением, в котором `'type': 'book', 'api' : 2`, а ключу `'authors'` соответствует последовательность. Вернуть элементы последовательности в виде нового списка.

❷ Сопоставить с любым отображением, в котором `'type': 'book', 'api' : 1`, а ключу `'author'` соответствует произвольный объект. Вернуть этот объект в виде элемента списка.

❸ Любое другое отображение, в котором `'type': 'book'`, недопустимо, возбудить исключение `ValueError`.

❹ Сопоставить с любым отображением, в котором `'type': 'movie'`, а ключу `'director'` соответствует одиночный объект. Вернуть этот объект в виде элемента списка.

❺ Любой другой субъект недопустим, возбудить исключение `ValueError`.

В примере 3.2 продемонстрированы полезные приемы обработки слабоструктурированных данных, в частности записей в формате JSON:

- включить поле, описывающее вид записи (например, `'type': 'movie'`);
- включить поле, идентифицирующее версию схемы (например, `'api': 2'`), чтобы в будущем можно было модифицировать открытый API;
- предусмотреть ветви `case` для обработки недопустимых записей (например, `'book'`), а также перехватывающую ветвь.

Теперь посмотрим, как функция `get_creators` обрабатывает некоторые тесты.

```
>>> b1 = dict(api=1, author='Douglas Hofstadter',
...           type='book', title='Gödel, Escher, Bach')
>>> get_creators(b1)
['Douglas Hofstadter']
>>> from collections import OrderedDict
>>> b2 = OrderedDict(api=2, type='book',
...           title='Python in a Nutshell',
...           authors='Martelli Ravenscroft Holden'.split())
>>> get_creators(b2)
['Martelli', 'Ravenscroft', 'Holden']
>>> get_creators({'type': 'book', 'pages': 770})
Traceback (most recent call last):
    ...
ValueError: Invalid 'book' record: {'type': 'book', 'pages': 770}
>>> get_creators('Spam, spam, spam')
Traceback (most recent call last):
    ...
ValueError: Invalid record: 'Spam, spam, spam'
```

Заметим, что порядок ключей в образцах не играет роли, даже если субъект имеет тип `OrderedDict`, как в случае `b2`.

В отличие от последовательностей-образцов, сопоставление с отображениями-образцами считается успешным даже в случае частичного совпадения. В тестах субъекты `b1` и `b2` включают ключ `'title'`, отсутствующий в образце `'book'`, и тем не менее сопоставление успешно.

Использовать аргумент `**extra` для сопоставления с лишними парами ключ-значение необязательно, но если вы хотите собрать их в словарь, то можете указать одну переменную с префиксом `**`. Она должна быть последней в образце, а конструкция `**_` запрещена ввиду своей избыточности. Вот простой пример:

```
>>> food = dict(category='ice cream', flavor='vanilla', cost=199)
>>> match food:
...     case {'category': 'ice cream', **details}:
...         print(f'Ice cream details: {details}')
...
Ice cream details: {'flavor': 'vanilla', 'cost': 199}
```

В разделе «Автоматическая обработка отсутствующих ключей» ниже мы будем рассматривать тип `defaultdict` и другие отображения, для которых поиск ключа с помощью метода `__getitem__` (т. е. `d[key]`) всегда завершается успешно, потому что отсутствующие элементы создаются динамически. В контексте сопоставления с образцом сопоставление считается успешным, только если в субъекте уже присутствовали необходимые ключи до выполнения `match`.

Автоматическая обработка отсутствующих ключей не активируется, потому что механизм сопоставления с образцом всегда вызывает метод `d.get(key, sentinel)`, где `sentinel` по умолчанию является специальным маркером, который не может встретиться в пользовательских данных.

Разобравшись с синтаксисом и структурой, перейдем к изучению API отображений.

Стандартный API типов отображений

Модуль `collections.abc` содержит абстрактные базовые классы `Mapping` и `MutableMapping`, формализующие интерфейсы типа `dict` и родственных ему. См. рис. 3.1.

Основная ценность ABC – документирование и формализация минимального интерфейса отображений, а также использование в тестах с помощью функции `isinstance` в тех программах, которые должны поддерживать произвольные отображения:

```
>>> my_dict = {}
>>> isinstance(my_dict, abc.Mapping)
True
>>> isinstance(my_dict, abc.MutableMapping)
True
```

Использовать `isinstance` совместно с ABC зачастую лучше, чем проверять, принадлежит ли аргумент функции конкретному типу `dict`, потому что так можно использовать и другие типы отображений. Мы подробно обсудим этот вопрос в главе 13.

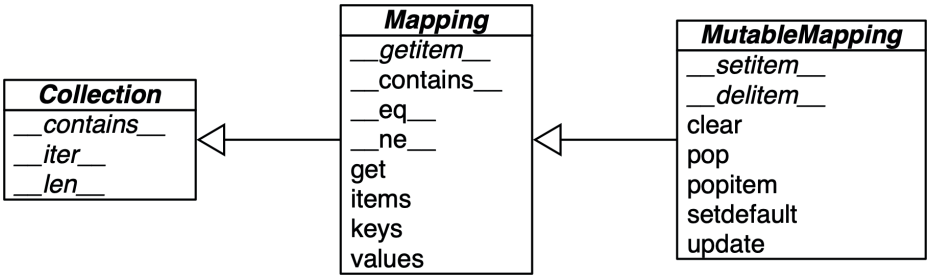


Рис. 3.1. Упрощенная UML-диаграмма класса `MutableMapping` и его суперклассов из модуля `collections.abc` (стрелки ведут от подклассов к суперклассам, курсивом набраны имена абстрактных классов и абстрактных методов)

Чтобы реализовать свое отображение, проще расширить класс `collections.UserDict` или обернуть `dict` с помощью композиции, чем создавать подклассы этих ABC. Класс `collections.UserDict` и все конкретные классы отображений в стандартной библиотеке инкапсулируют базовый словарь `dict`, который, в свою очередь, основан на хеш-таблице. Поэтому у всех них есть общее ограничение: ключи должны быть *хешируемыми* (к значениям это требование не относится). В следующем разделе объясняется, что это означает.

Что значит «хешируемый»?

Вот часть определения хешируемости, взятая из глоссария Python (*https://docs.python.org/3/glossary.html#term-hashable*):

> Объект называется хешируемым, если имеет хеш-код, который не изменяется на протяжении всего времени его жизни (у него должен быть метод `__hash__()`), и допускает сравнение с другими объектами (у него дол-

жен быть метод `__eq__()`). Если в результате сравнения хешируемых объектов оказывается, что они равны, то и их хеш-коды должны быть равны[1].

Все числовые типы и плоские неизменяемые типы `str` и `bytes` являются хешируемыми. Объект типа `frozenset` всегда хешируемый, потому что его элементы должны быть хешируемыми по определению. Объект типа `tuple` является хешируемым только тогда, которые хешируемы все его элементы. Взгляните на кортежи `tt`, `tl` и `tf`:

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> tl = (1, 2, [30, 40])
>>> hash(tl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```

Хеш-код объекта может зависеть от версии Python, машинной архитектуры и *начального значения*, включенного в процесс вычисления хешей из соображений безопасности[2]. Хеш-код корректно реализованного объекта является гарантированно постоянным только в рамках одного процесса.

Любой пользовательский тип является хешируемым по определению, потому что его хеш-значение равно `id()`, а метод `__eq__()`, унаследованный от класса `object`, просто сравнивает идентификаторы объектов. Если объект реализует пользовательский метод `__eq__()`, учитывающий внутреннее состояние, то он будет хешируемым, только если его метод `__hash__()` всегда возвращает один и тот же хеш-код. На практике это требование означает, что методы `__eq__()` и `__hash__()` должны принимать во внимание только те атрибуты экземпляра, которые не изменяются на протяжении всей жизни объекта.

Теперь обсудим API наиболее употребительных типов отображений в Python: `dict`, `defaultdict` и `OrderedDict`.

Обзор наиболее употребительных методов отображений

Базовый API отображений очень хорошо развит. В табл. 3.1 показаны методы, реализованные в классе `dict` и двух его самых полезных разновидностях: `defaultdict` и `OrderedDict` (тот и другой определены в модуле `collections`).

[1] В глоссарии Python (*https://docs.python.org/3/glossary.html#term-hashable*) употребляется термин «хеш-значение», а не «хеш-код». Я предпочитаю «хеш-код», потому что это понятие часто встречается в контексте отображений, элементы которых состоят из ключей и значений, и, следовательно, одновременное употребление терминов «хеш-значение» и «значение» могло бы привести к путанице. В этой книге используется только «хеш-код».

[2] О проблемах безопасности и принятых решениях см. документ PEP 456 «Secure and interchangeable hash algorithm» (*https://peps.python.org/pep-0456/*).

Таблица 3.1. Методы типов отображений types dict, collections.defaultdict и collections.OrderedDict (для краткости методы, унаследованные от object, опущены); необязательные аргументы заключены в квадратные скобки

	dict	defaultdict	OrderedDict	
`d.clear()`	●	●	●	Удаление всех элементов
`d.__contains__(k)`	●	●	●	`k` входит в `d`
`d.copy()`	●	●	●	Поверхностная копия
`d.__copy__()`		●		Поддержка `copy.copy`
`d.default_factory`		●		Вызываемый объект, к которому обращается метод `__missing__` в случае отсутствия значения[a]
`s.__delitem__(p)`	●	●	●	`del d[k]` – удаление элемента с ключом `k`
`d.fromkeys(itm [initial])`	●	●	●	Новое отображение, ключи которого поставляет итерируемый объект, и с необязательным начальным значением (по умолчанию `None`)
`d.get(k, [default])`	●	●	●	Получить элемент с ключом `k`, а если такой ключ отсутствует, вернуть `default` или `None`
`d.__getitem__(k)`	●	●	●	`d[k]` – получить элемент с ключом `k`
`d.items()`	●	●	●	Получить *представление* элементов – множество пары `(key, value)`
`d.__iter__()`	●	●	●	Получение итератора по ключам
`d.keys()`	●	●	●	Получить *представление* ключей
`d.__len__()`	●	●	●	`len(d)` – количество элементов
`d.__missing__(k)`		●		Вызывается, когда `__getitem__` не может найти элемент
`d.move_to_end(k, [last])`			●	Переместить ключ `k` в первую или последнюю позицию (`last` по умолчанию равно `True`)
`d.__or__(other)`	●	●	●	Поддерживает операцию `d1 | d2`, создающую объединение `d1` и `d2` (Python ⩾ 3.9)
`d.__ior__(other)`	●	●	●	Поддерживает операцию `d1 |= d2`, добавляющую в `d1` содержимое `d2` (Python ⩾ 3.9)
`d.pop(k, [default])`	●	●	●	Удалить и вернуть значение с ключом `k`, а если такой ключ отсутствует, вернуть `default` или `None`
`d.popitem()`	●	●	●	Удалить и вернуть произвольный элемент `(key, value)`[b]

Окончание табл. 3.1

	dict	defaultdict	OrderedDict	
`d.__reversed__()`	●	●	●	Получить итератор для перебора ключей от последнего к первому вставленному
`d.__ror__(other)`	●	●	●	Поддерживает операцию `other | d` – оператор инверсного объединения (Python ⩾ 3.9)[c]
`d.setdefault(k, [default])`	●	●	●	Если `k` принадлежит `d`, вернуть `d[k]`, иначе положить `d[k] = default` и вернуть это значение
`d.__setitem__(k, v)`	●	●	●	`d[k] = v` – поместить `v` в элемент с ключом `k`
`d.update(m, [**kargs])`	●	●	●	Обновить `d` элементами из отображения или итерируемого объекта, возвращающего пары `(key, value)`
`d.values()`	●	●	●	Получить *представление* значений

[a] `default_factory` – не метод, а атрибут – вызываемый объект, задаваемый пользователем при создании объекта `defaultdict`.

[b] Метод `OrderedDict.popitem()` удаляет первый вставленный элемент (дисциплина FIFO); если необязательный аргумент `last` равен `True`, то удаляет последний вставленный элемент (дисциплина LIFO).

[c] Инверсные операторы обсуждаются в главе 16.

То, как метод `d.update(m)` трактует свой первый аргумент `m`, – яркий пример *утиной типизации* (duck typing): сначала проверяется, есть ли у `m` метод `keys`, и если да, то предполагается, что это отображение. В противном случае `update()` производит обход `m` в предположении, что элементами являются пары `(key, value)`. Конструкторы большинства отображений в Python применяют логику метода `update()`, а значит, отображение можно инициализировать как другим отображением, так и произвольным итерируемым объектом, порождающим пары `(key, value)`.

Метод `setdefault` – тонкая штучка. Нужен он не всегда, но, когда нужен, позволяет существенно ускорить работу, избегая излишних операций поиска ключа. В следующем разделе на практическом примере объясняется, как им пользоваться.

Вставка и обновление изменяемых значений

В полном соответствии с философией *быстрого отказа* доступ к словарю `dict` с помощью конструкции `d[k]` возбуждает исключение, если ключ `k` отсутствует. Любой питонист знает об альтернативной конструкции `d.get(k, default)`, которая применяется вместо `d[k]`, если иметь значение по умолчанию удобнее, чем обрабатывать исключение `KeyError`. Однако если нужно обновить изменяемое значение, то есть способ лучше.

Рассмотрим скрипт для индексирования текста, порождающий отображение, в котором ключом является слово, а значением – список позиций, в которых это слово встречается (см. пример 3.3).

Пример 3.3. Частичный результат работы скрипта 3.4 применительно к обработке текста «Дзен Python»; в каждой строке показано слово и список его вхождений, представленных парами (номер_строки, номер-столбца)

```
$ python3 index0.py zen.txt
a [(19, 48), (20, 53)]
Although [(11, 1), (16, 1), (18, 1)]
ambiguity [(14, 16)]
and [(15, 23)]
are [(21, 12)]
aren [(10, 15)]
at [(16, 38)]
bad [(19, 50)]
be [(15, 14), (16, 27), (20, 50)]
beats [(11, 23)]
Beautiful [(3, 1)]
better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11), (17, 8), (18, 25)]
...
```

В примере 3.4 показан неоптимальный скрипт, демонстрирующий одну ситуацию, когда `dict.get` – не лучший способ обработки отсутствия ключа. Он основан на примере Алекса Мартелли[1].

Пример 3.4. index0.py: применение метода `dict.get` для выборки и обновления списка вхождений слова в индекс (в примере 3.4 показано лучшее решение)

```
"""Строит индекс, отображающий слово на список его вхождений"""

import re
import sys

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            # некрасиво; написано только для демонстрации идеи
            occurrences = index.get(word, []) ❶
            occurrences.append(location)      ❷
            index[word] = occurrences         ❸

# напечатать в алфавитном порядке
for word in sorted(index, key=str.upper):     ❹
    print(word, index[word])
```

❶ Получить список вхождений слова `word` или `[]`, если оно не найдено.
❷ Добавить новое вхождение в `occurrences`.
❸ Поместить модифицированный список `occurrences` в словарь `dict`; при этом производится второй поиск в индексе.

[1] Оригинальный скрипт представлен на слайде 41 презентации Мартелли «Учим Python заново» (*http://www.aleax.it/Python/accu04_Relearn_Python_alex.pdf*). Его скрипт демонстрирует использование `dict.setdefault`, показанное в примере 3.5.

❹ При задании аргумента `key` функции `sorted` мы не вызываем `str.upper`, а только передаем ссылку на этот метод, чтобы `sorted` могла нормализовать слова перед сортировкой[1].

Три строчки, относящиеся к обработке `occurrences` в примере 3.4, можно заменить одной, воспользовавшись методом `dict.setdefault`. Пример 3.5 ближе к оригинальному коду Алекса Мартелли.

Пример 3.5. index.py: применение метода `dict.setdefault` для выборки и обновления списка вхождений слова в индекс; в отличие от примера 3.4, понадобилась только одна строчка

```
«»»Строит индекс, отображающий слово на список его вхождений»»»

import re
import sys

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            index.setdefault(word, []).append(location) ❶

# напечатать в алфавитном порядке
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

❶ Получить список вхождений слова `word` или установить его равным `[]`, если слово не найдено; `setdefault` возвращает значение, поэтому список можно обновить без повторного поиска.

Иными словами, строка...

```
my_dict.setdefault(key, []).append(new_value)
```

... дает такой же результат, как ...

```
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value)
```

... с тем отличием, что во втором фрагменте производится по меньшей мере два поиска ключа (три, если ключ не найден), тогда как `setdefault` довольствуется единственным поиском.

Смежный вопрос – обработка отсутствия ключа при любом поиске (а не только при вставке) – тема следующего раздела.

[1] Здесь мы видим пример использования метода в качестве полноправной функции, подробнее эта тема обсуждается в главе 7.

Автоматическая обработка отсутствующих ключей

Иногда удобно, чтобы отображение возвращало некоторое специальное значение, если искомый ключ отсутствует. К решению данной задачи есть два подхода: первый – использовать класс `defaultdict` вместо `dict`, второй – создать подкласс `dict` или любого другого типа отображения и добавить метод `__missing__`. Ниже рассматриваются оба способа.

defaultdict: еще один подход к обработке отсутствия ключа

Экземпляр класса `collections.defaultdict` создает элементы, имеющие значение по умолчанию, динамически, если при использовании конструкции `d[k]` ключ не был найден. В примере 3.6 предлагается еще одно элегантное решение задачи индексирования текста из примера 3.5.

Работает это следующим образом: при конструировании объекта `defaultdict` задается вызываемый объект, который порождает значение по умолчанию всякий раз, как методу `__getitem__` передается ключ, отсутствующий в словаре.

Например, пусть `defaultdict` создан как `dd = defaultdict(list)`. Тогда если ключ `'new-key'` отсутствует в `dd`, то при вычислении выражения `dd['new-key']` выполняются следующие действия:

1. Вызвать `list()` для создания нового списка.
2. Вставить список в `dd` в качестве значения ключа `'new-key'`.
3. Вернуть ссылку на этот список.

Вызываемый объект, порождающий значения по умолчанию, хранится в атрибуте экземпляра `default_factory`.

Пример 3.6. index_default.py: использование экземпляра `defaultdict` вместо метода `setdefault`

```
"""Строит индекс, отображающий слово на список его вхождений"""

import collections
import re
import sys

WORD_RE = re.compile(r'\w+')

index = collections.defaultdict(list) ❶
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            index[word].append(location) ❷

# напечатать в алфавитном порядке
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

❶ Создать `defaultdict`, задав в качестве `default_factory` конструктор `list`.

❷ Если слова `word` еще нет в `index`, то вызывается функция `default_factory`, кото-

рая порождает отсутствующее значение – в данном случае пустой список. Это значение присваивается `index[word]` и возвращается, так что операция `.append(location)` всегда завершается успешно.

Если атрибут `default_factory` не задан, то в случае отсутствия ключа, как обычно, возбуждается исключение `KeyError`.

Атрибут `default_factory` объекта `defaultdict` вызывается только для того, чтобы предоставить значение по умолчанию при обращении к методу `__getitem__` и только к нему. Например, если `dd` – объект класса `defaultdict` и `k` – отсутствующий ключ, то при вычислении выражения `dd[k]` происходит обращение к `default_factory` для создания значения по умолчанию, а вызов `dd.get(k)` все равно возвращает `None` и `k in dd` равно `False`.

А почему `defaultdict` обращается к `default_factory`? Всему виной специальный метод `__missing__`. Его мы и обсудим далее.

Метод __missing__

В основе механизма обработки отсутствия ключей в отображениях лежит метод, которому как нельзя лучше подходит имя `__missing__`. Он не определен в базовом классе `dict`, но `dict` знает о нем: если создать подкласс `dict` и реализовать в нем метод `__missing__`, то стандартный метод `dict.__getitem__` будет обращаться к нему всякий раз, как не найдет ключ, – вместо того чтобы возбуждать исключение `KeyError`.

Допустим, нам нужно отображение, в котором ключ перед поиском преобразуется в тип `str`. Конкретный пример дает библиотека работы с устройствами для IoT[1], в которой программируемая плата с контактами ввода-вывода общего назначения (например, Raspberry Pi или Arduino) представлена объектом класса `Board` с атрибутом `my_board.pins`, который является отображением идентификаторов физических контактов на программные объекты контактов. Идентификатор физического контакта может задаваться числом или строкой вида `"A0"` либо `"P9_12"`. Для единообразия желательно, чтобы все ключи `board.pins` были строками, но хорошо бы, чтобы и обращение вида `my_arduino.pin[13]` тоже работало, тогда начинающие не будут впадать в ступор, желая зажечь светодиод, подключенный к контакту 13 на плате Arduino. В примере 3.7 показано, как такое отображение могло бы быть реализовано.

Пример 3.7. При поиске по нестроковому ключу объект `StrKeyDict0` преобразует его в тип `str` в случае отсутствия

```
Tests for item retrieval using `d[key]` notation::

    >>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
    >>> d['2']
    'two'
    >>> d[4]
    'four'
```

[1] Одна из таких библиотек – *Pingo.io* (*https://github.com/pingo-io/pingo-py*), развитие которой прекращено.

```
    >>> d[1]
    Traceback (most recent call last):
        ...
    KeyError: '1'

Tests for item retrieval using `d.get(key)` notation::

    >>> d.get('2')
    'two'
    >>> d.get(4)
    'four'
    >>> d.get(1, 'N/A')
    'N/A'

Tests for the `in` operator::

    >>> 2 in d
    True
    >>> 1 in d
    False
```

В примере 3.8 реализован класс `StrKeyDict0`, для которого все приведенные выше тесты проходят.

Более правильный способ реализовать тип пользовательского отображения – унаследовать классу `collections.UserDict`, а не `dict` (мы так и поступим в примере 3.9). Здесь мы создали подкласс `dict` просто для демонстрации того, что метод `__missing__` поддерживается встроенным методом `dict.__getitem__`.

Пример 3.8. Класс `StrKeyDict0` преобразует нестроковые ключи в тип `str` во время поиска (см. тесты в примере 3.7)

```
class StrKeyDict0(dict): ❶

    def __missing__(self, key): ❷
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)] ❸

    def get(self, key, default=None):
        try:
            return self[key] ❹
        except KeyError:
            return default ❺

    def __contains__(self, key):
        return key in self.keys() or str(key) in self.keys() ❻
```

❶ `StrKeyDict0` наследует `dict`.

❷ Проверить, принадлежит ли ключ `key` типу `str`. Если да и при этом отсутствует в словаре, возбудить исключение `KeyError`.

❸ Преобразовать `key` в `str` и искать.

❹ Метод `get` делегирует свою работу методу `__getitem__` благодаря нотации `self[key]`; это приводит в действие наш метод `__missing__`.

❺ Если возникло исключение `KeyError`, значит, метод `__missing__` уже завершился с ошибкой, поэтому вернуть `default`.
❻ Искать сначала по немодифицированному ключу (экземпляр может содержать нестроковые ключи), а затем по строке `str`, построенной по ключу.

Задайтесь вопросом, зачем в реализации `__missing__` необходима проверка `isinstance(key, str)`.

Без этой проверки наш метод `__missing__` работал бы для любого ключа `k` – не важно, принадлежит он типу `str` или нет, – если только `str(k)` порождает существующий ключ. Но если ключ `str(k)` не существует, то возникла бы бесконечная рекурсия. В последней строке вычисление `self[str(key)]` привело бы к вызову `__getitem__` с параметром, равным строковому представлению ключа, а это, в свою очередь, – снова к вызову `__missing__`.

Метод `__contains__` в этом примере также необходим для обеспечения согласованного поведения, потому что его вызывает операция `k in d`, однако реализация данного метода, унаследованная от `dict`, не обращается к `__missing__` в случае отсутствия ключа. В нашей реализации `__contains__` есть тонкий нюанс: мы не проверяем наличие ключа принятым в Python способом – `k in my_dict` – потому что конструкция `str(key) in self` привела бы к рекурсивному вызову `__contains__`. Чтобы избежать этого, мы явно ищем ключ в `self.keys()`.

Поиск вида `k in my_dict.keys()` эффективен в Python 3 даже для очень больших отображений, потому что `dict.keys()` возвращает представление, похожее на множество, как мы увидим в разделе «Теоретико-множественные операции над представлениями словарей» ниже. Однако не забывайте, что `k in my_dict` делает то же самое, причем быстрее, потому что не нужно искать среди атрибутов метод `.keys()`.

У меня была причина использовать `self.keys()` в методе `__contains__` в примере 3.8. Проверка наличия немодифицированного ключа – `key in self.keys()` – необходима для корректности, потому что класс `StrKeyDict0` не гарантирует, что все ключи словаря обязательно имеют тип `str`. Наша цель состоит только в том, чтобы сделать поиск более дружелюбным, а не навязывать пользователю типы.

Определенные пользователем классы, производные от отображений из стандартной библиотеки, могут использовать или не использовать метод `__missing__` в качестве запасного варианта в собственной реализации `__getitem__`, `get` или `__contains__`. Мы обсудим это в следующем разделе.

Несогласованное использование __missing__ в стандартной библиотеке

Рассмотрим следующие сценарии и то, как в них обрабатывается поиск отсутствующего ключа.

Подкласс `dict`

> Подкласс `dict`, в котором реализован только метод `__missing__` и никаких других. В этом случае `__missing__` может вызываться лишь при использовании конструкции `d[k]`, которая обращается к методу `__getitem__`, унаследованному от `dict`.

Подкласс `collections.UserDict`

Подкласс `UserDict`, в котором реализован только метод `__missing__` и никаких других. В этом случае метод `get`, унаследованный от `UserDict`, вызывает `__getitem__`. Это значит, что `__missing__` может вызываться для обработки поиска с помощью конструкций `d[k]` или `d.get(k)`.

Подкласс `abc.Mapping` *с простейшим* `__getitem__`

Минимальный подкласс `abc.Mapping`, в котором реализован метод `__missing__` и необходимые абстрактные методы, в т. ч. имеется реализация `__getitem__`, не обращающаяся к `__missing__`. В таком классе метод `__missing__` никогда не вызывается.

Подкласс `abc.Mapping` *с* `__getitem__`*, вызывающим* `__missing__`

Минимальный подкласс `abc.Mapping`, в котором реализован метод `__missing__` и необходимые абстрактные методы, в т. ч. имеется реализация `__getitem__`, обращающаяся к `__missing__`. В таком классе метод `__missing__` вызывается, когда поиск производится с помощью конструкций `d[k]`, `d.get(k)` и `k in d`.

В скрипте *missing.py* (*https://github.com/fluentpython/example-code-2e/blob/master/03-dict-set/missing.py*) в репозитории кода приведены демонстрации всех описанных выше сценариев.

Во всех четырех сценариях предполагаются минимальные реализации. Если ваш подкласс реализует методы `__getitem__`, `get` и `__contains__`, то использовать в них `__missing__` или нет, зависит от конкретных потребностей. Цель этого раздела – показать, что при создании подклассов стандартных библиотечных отображений, в которых предполагается использовать `__missing__`, необходима осторожность, потому что базовые классы по умолчанию поддерживают разные поведения.

Не забывайте, что поведение методов `setdefault` и `update` тоже зависит от поиска ключа. И наконец, в зависимости от логики реализации `__missing__` в своем классе вы, возможно, должны будете реализовать специальную логику в `__setitem__`, чтобы избежать несогласованного или неожиданного поведения. Пример такого рода мы встретим в разделе «Создание подкласса UserDict вместо dict».

До сих пор мы рассматривали типы отображений `dict` и `defaultdict`, но в стандартной библиотеке имеются и другие реализации отображения. Обсудим их.

Вариации на тему dict

В этом разделе мы дадим обзор типов отображений, включенных в модуль стандартной библиотеки `collections` (помимо рассмотренного выше `defaultdict`):

collections.OrderedDict

Поскольку начиная с версии Python 3.6 встроенный словарь `dict` также хранит ключи упорядоченными, использовать `OrderedDict` имеет смысл главным образом для поддержания обратной совместимости с предыдущими версиями. Тем не менее в документации приводится несколько оставшихся различий между

dict и OrderedDict, которые я повторю здесь, перечислив в порядке, отвечающем частоте использования в повседневной практике.

- Оператор равенства в классе OrderedDict проверяет, что порядок следования ключей одинаков.
- Метод popitem() в классе OrderedDict имеет другую сигнатуру. Он принимает необязательный аргумент, указывающий, какой элемент извлечь.
- В классе OrderedDict имеется метод move_to_end(), который эффективно переходит к последнему элементу в словаре.
- При проектировании обычного dict на первое место ставилась высокая эффективность операций отображения, а отслеживание порядка вставки было вторичным.
- При проектировании OrderedDict на первое место ставилась высокая эффективность операций переупорядочения, а эффективность использования памяти, скорость итерирования и производительность операций обновления были вторичны.
- Алгоритмически OrderedDict справляется с частыми операциями переупорядочения лучше, чем dict. Поэтому он подходит для отслеживания недавних операций доступа (например, в LRU-кеше).

collections.ChainMap

Хранит список отображений, так что их можно просматривать как единое целое. Поиск производится в каждом отображении в порядке их перечисления в конструкторе и завершается успешно, если ключ найден хотя бы в одном. Например:

```
>>> d1 = dict(a=1, b=3)
>>> d2 = dict(a=2, b=4, c=6)
>>> from collections import ChainMap
>>> chain = ChainMap(d1, d2)
>>> chain['a']
1
>>> chain['c']
6
```

Экземпляр ChainMap не копирует входные отображения, а хранит ссылки на них. Все модификации и вставки ChainMap применяются только к первому из входных отображений. Продолжим предыдущий пример:

```
>>> chain['c'] = -1
>>> d1
{'a': 1, 'b': 3, 'c': -1}
>>> d2
{'a': 2, 'b': 4, 'c': 6}
```

ChainMap полезен при реализации интерпретаторов языков с вложенными областями видимости, когда каждая область видимости представлена отдельным отображением, в направлении от самого внутреннего к самому внешнему. В разделе «Объекты ChainMap» документации по модулю collections (*https://docs.python.org/3/library/collections.html#collections.ChainMap*) есть несколько примеров использования ChainMap, включая и следующий фрагмент, иллюстрирующий базовые правила поиска имен переменных в Python:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

В примере 18.14 показано, как подкласс `ChainMap` применяется в интерпретаторе подмножества языка Scheme.

collections.Counter

Отображение, в котором с каждым ключом ассоциирован счетчик. Обновление существующего ключа увеличивает его счетчик. Этот класс можно использовать для подсчета количества хешируемых объектов или в качестве мультимножества (обсуждается ниже в данном разделе). В классе `Counter` реализованы операторы `+` и `-` для объединения серий и другие полезные методы, например `most_common([n])`, который возвращает упорядоченный список кортежей, содержащий *n* самых часто встречающихся элементов вместе с их счетчиками; документацию см. по адресу *https://docs.python.org/3/library/collections.html#collections.Counter*. Ниже демонстрируется применение `Counter` для подсчета числа различных букв в слове:

```
>>> ct = collections.Counter('abracadabra')
>>> ct
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.update('aaaaazzz')
>>> ct
Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.most_common(3)
[('a', 10), ('z', 3), ('b', 2)]
```

Обратите внимание, что оба ключа `'b'` и `'r'` делят третье место, но `ct.most_common(3)` показывает только три счетчика.

Чтобы воспользоваться классом `collections.Counter` как мультимножеством, представьте, что каждый ключ – элемент множества, а счетчик – количество вхождений этого элемента в множество.

shelve.Shelf

Модуль `shelve` из стандартной библиотеки предоставляет постоянное хранилище для отображения строковых ключей на объекты Python, сериализованные в двоичном формате `pickle`. Название `shelve` (полка) – намек на то, что банки с соленьями (pickle) хранятся на полках.

Функция уровня модуля `shelve.open` возвращает экземпляр `shelve.Shelf` – простую базу ключей и значений, поддерживаемую модулем `dbm` и обладающую следующими свойствами:

- `shelve.Shelf` является подклассом `abc.MutableMapping`, т. е. предоставляет основные методы, ожидаемые от типа отображения;
- кроме того, `shelve.Shelf` предоставляет еще несколько методов для управления вводом-выводом, например `sync` и `close`;
- экземпляр `Shelf` является контекстным менеджером, поэтому его можно включать в блок `with`, гарантирующий закрытие после использования;
- ключи и значения сохраняются при каждом присваивании нового значения ключу. Ключи должны быть строками;
- значения должны быть объектами, которые модуль `pickle` умеет сериализовывать.

В документации по модулям shelve (*https://docs.python.org/3/library/shelve.html*), dbm (*https://docs.python.org/3/library/ dbm.html*) и pickle (*https://docs.python.org/3/library/pickle.html*) приведены дополнительные сведения и описаны некоторые подводные камни.

Модуль `pickle` легко использовать в простых случаях, но у него есть несколько недостатков. Прочитайте статью Нэда Бэтчелдера «Pickle's nine flaws» (*https://nedbatchelder.com/blog/202006/pickles_nine_flaws.html*), прежде чем принимать любое решение, подразумевающее использование `pickle`. Кстати, Нэд упоминает и о других форматах сериализации, на которые стоит обратить внимание.

Классы `OrderedDict`, `ChainMap`, `Counter` и `Shelf` готовы к использованию, но при желании их поведение можно модифицировать, создав подклассы. С другой стороны, `UserDict` – базовый класс, специально предназначенный для расширения.

Создание подкласса UserDict вместо dict

Рекомендуется создавать новый тип отображения путем расширения класса `collections.UserDict`, а не `dict`. Мы убедимся в этом, попытавшись расширить класс `StrKeyDict0` из примера 3.8, так чтобы любой ключ, добавляемый в отображение, сохранялся в виде строки `str`.

Основная причина, по которой предпочтительнее наследовать классу `UserDict`, а не `dict`, заключается в том, что в реализации `dict` некоторые углы срезаны, что вынуждает нас переопределять методы, которые можно без всяких проблем унаследовать от `UserDict`[1].

Отметим, что `UserDict` не наследует `dict`, а пользуется композицией: хранит в атрибуте `data` экземпляр `dict`, где и находятся сами элементы. Это позволяет избежать нежелательной рекурсии при кодировании таких специальных методов, как `__setitem__`, и упрощает код `__contains__` по сравнению с тем, что показан в примере 3.8.

Благодаря `UserDict` класс `StrKeyDict` (пример 3.9) получился короче, чем `StrKeyDict0` (пример 3.8), но умеет при этом больше: он хранит все ключи в виде `str`, обходя тем самым неприятные сюрпризы, возможные, если при создании или обновлении экземпляра были добавлены данные с нестроковыми ключами.

Пример 3.9. `StrKeyDict` всегда преобразует нестроковые ключи в тип `str` – при вставке, обновлении и поиске

```
import collections

class StrKeyDict(collections.UserDict): ❶

    def __missing__(self, key): ❷
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)]
```

[1] Точное описание проблем, сопряженных с наследованием `dict` и другим встроенным классам, см. в разделе «Сложности наследования встроенным типам» главы 14.

```
    def __contains__(self, key):
        return str(key) in self.data ❸

    def __setitem__(self, key, item):
        self.data[str(key)] = item ❹
```

❶ `StrKeyDict` расширяет `UserDict`.
❷ Метод `__missing__` точно такой же, как в примере 3.8.
❸ Метод `__contains__` проще: можно предполагать, что все хранимые ключи имеют тип `str`, так что можно искать ключ в самом словаре `self.data`, а не вызывать `self.keys()`, как в классе `StrKeyDict0`.
❹ Метод `__setitem__` преобразует любой ключ в тип `str`. Этот метод проще переопределить, если можно делегировать работу атрибуту `self.data`.

Поскольку `UserDict` – подкласс `MutableMapping`, остальные методы, благодаря которым `StrKeyDict` является полноценным отображением, наследуются от `UserDict`, `MutableMapping` или `Mapping`. В двух последних есть несколько полезных конкретных методов, хотя они и являются абстрактными базовыми классами (ABC). Стоит отметить следующие методы.

`MutableMapping.update`

Этот метод можно вызывать напрямую, но им также пользуется метод `__init__` для инициализации экземпляра другими отображениями, итерируемыми объектами, порождающими пары `(key, value)`, и именованными аргументами. Поскольку для добавления элементов он использует конструкцию `self[key] = value`, то в конечном итоге будет вызвана наша реализация `__setitem__`.

`Mapping.get`

В классе `StrKeyDict0` (пример 3.8) мы вынуждены были самостоятельно написать метод `get`, чтобы получаемые результаты были согласованы с `__getitem__`, но в примере 3.9 мы унаследовали `Mapping.get`, который реализован в точности так, как `StrKeyDict0.get` (см. исходный код Python (*https://github.com/python/cpython/blob/0bbf30e2b910bc9c5899134ae9d73a8df968da35/Lib/_collections_abc.py#L813*)).

Уже написав класс `StrKeyDict`, я обнаружил, что Антуан Питру (Antoine Pitrou) опубликовал документ PEP 455 «Adding a key-transforming dictionary to collections» (*https://www.python.org/dev/peps/pep-0455/*) и исправление, дополняющее модуль `collections` классом `TransformDict`. Он более общий, чем `StrKeyDict`, и сохраняет ключи в том виде, в котором они переданы, прежде чем применить к ним преобразование. Предложение PEP 455 было отвергнуто в мае 2015 года – см. возражение Раймонда Хэттингера (*https://mail.python.org/pipermail/python-dev/2015-May/140003.html*). Чтобы поэкспериментировать с классом `TransformDict`, я «выдернул» заплату Питру из проблемы 18986 (*https://github.com/fluentpython/example-code-2e*) в отдельный модуль (*03-dictset/transformdict.py*) в репозитории кода к этой книге (*https://github.com/fluentpython/example-code-2e*)).

Мы знаем, что существует несколько неизменяемых типов последовательностей, а как насчет неизменяемого словаря? В стандартной библиотеке такого не имеется, но выход есть. Читайте дальше.

Неизменяемые отображения

Все типы отображений в стандартной библиотеке изменяемые, но иногда нужно гарантировать, что пользователь не сможет по ошибке модифицировать отображение. Конкретный пример снова дает проект *Pingo*, который я уже описывал в разделе «Метод `__missing__`» выше: отображение `board.pins` представляет физические контакты GPIO на плате. Поэтому было бы желательно предотвратить непреднамеренное изменение `board.pins`, потому что нельзя же изменять оборудование с помощью программы, т. е. любая такая модификация оказалась бы не согласованной с физическим устройством.

Модуль `types` содержит класс-обертку `MappingProxyType`, который получает отображение и возвращает объект `mappingproxy`, допускающий только чтение, но при этом являющийся динамическим представлением исходного отображения. Это означает, что любые изменения исходного отображения будут видны и в `mappingproxy`, но через него такие изменения сделать нельзя. Демонстрация приведена в примере 3.10.

Пример 3.10. Класс `MappingProxyType` строит по словарю объект `mappingproxy`, допускающий только чтение

```
>>> d = {1: 'A'}
>>> d_proxy = MappingProxyType(d)
>>> d_proxy
mappingproxy({1: 'A'})
>>> d_proxy[1] ❶
'A'
>>> d_proxy[2] = 'x' ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> d[2] = 'B'
>>> d_proxy ❸
mappingproxy({1: 'A', 2: 'B'})
>>> d_proxy[2]
'B'
>>>
```

❶ Элементы `d` можно видеть через `d_proxy`.
❷ Произвести изменения через `d_proxy` невозможно.
❸ Представление `d_proxy` динамическое: любое изменение сразу же отражается.

Вот как этим можно воспользоваться на практике в случае программирования оборудования: конструктор конкретного подкласса `Board` инициализирует закрытое отображение объектами, представляющими контакты, и раскрывает его клиентам API с помощью открытого атрибута `.pins`, реализованного как `mappingproxy`. Таким образом, клиент не сможет по ошибке добавлять, удалять и изменять контакты.

Далее мы рассмотрим представления, которые позволяют очень эффективно производить операции с `dict` без необходимости копировать данные.

Представления словаря

Такие методы `dict`, как `.keys()`, `.values()` и `.items()`, возвращают экземпляры классов `dict_keys`, `dict_values` и `dict_items` соответственно. Эти представления словаря – проекции внутренних структур данных, используемых в реализации `dict`, допускающие только чтение. Они обходятся без накладных расходов, присущих эквивалентным методам в Python 2, которые возвращали списки, дублирующие данные, уже хранящиеся в словаре. Кроме того, они заменяют старые методы, возвращавшие итераторы.

В примере 3.11 показаны некоторые простые операции, поддерживаемые всеми представлениями словарей.

Пример 3.11. Метод `.values()` возвращает представление значений в `dict`

```
>>> d = dict(a=10, b=20, c=30)
>>> values = d.values()
>>> values
dict_values([10, 20, 30]) ❶
>>> len(values) ❷
3
>>> list(values) ❸
[10, 20, 30]
>>> reversed(values) ❹
<dict_reversevalueiterator object at 0x10e9e7310>
>>> values[0] ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_values' object is not subscriptable
```

❶ Метод `repr` объекта представления показывает его содержимое.
❷ Можно запросить длину представления `len`.
❸ Представления допускают итерирование, поэтому из них легко создавать списки.
❹ Представления реализуют метод `__reversed__`, возвращающий пользовательский итератор.
❺ Для получения отдельного элемента представления нельзя использовать оператор `[]`.

Объект представления – это динамический прокси-объект. Если исходный словарь изменился, то изменения сразу же становятся видны через имеющееся представление. Продолжим пример 3.11.

```
>>> d['z'] = 99
>>> d
{'a': 10, 'b': 20, 'c': 30, 'z': 99}
>>> values
dict_values([10, 20, 30, 99])
```

Классы `dict_keys`, `dict_values` и `dict_items` внутренние: они недоступны через `__builtins__` или какой-либо модуль из стандартной библиотеки. Даже получив

ссылку на любой из них, вы не сможете воспользоваться ей в коде на Python, чтобы создать новое представление.

```
>>> values_class = type({}.values())
>>> v = values_class()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'dict_values' instances
```

Класс `dict_values` – простейшее из представлений словарей: он реализует только специальные методы `__len__`, `__iter__` и `__reversed__`. Классы `dict_keys` и `dict_items` дополнительно реализуют несколько методов множества – почти столько же, сколько класс `frozenset`. После того как мы рассмотрим множества, мы еще вернемся к классам `dict_keys` и `dict_items` в разделе «Теоретико-множественные операции над представлениями словарей».

А теперь, вооруженные знаниями о внутренней реализации `dict`, сформулируем некоторые правила и дадим рекомендации.

Практические последствия внутреннего устройства класса dict

Реализация класса Python `dict` на основе хеш-таблиц очень эффективна, но важно понимать, какие последствия вытекают из такого проектного решения.

- Ключи должны быть хешируемыми объектами. Они должны правильно реализовывать методы `__hash__` и `__eq__`, как описано в разделе «Что значит “хешируемый”?».
- Доступ к элементу по ключу производится очень быстро. Словарь может содержать миллионы ключей, но Python для нахождения ключа нужно только вычислить хеш-код ключа и получить по нему смещение относительно начала хеш-таблицы; возможно, еще придется выполнить несколько попыток, чтобы добраться до нужной записи.
- Порядок ключей сохраняется, это побочный эффект более компактного размещения словаря в памяти, реализованного в CPython 3.6 и ставшего официальной особенностью языка в версии 3.7.
- Несмотря на новое компактное размещение, словарям внутренне присущи большие накладные расходы на хранение в памяти. Самой компактной внутренней структурой данных для контейнера был бы массив указателей на элементы[1]. По сравнению с этим хеш-таблица должна хранить больше данных на каждую запись, причем хотя бы треть записей должна оставаться незаполненной, чтобы интерпретатор мог работать эффективно.
- Для экономии памяти избегайте создания атрибутов экземпляров вне метода `__init__`.

Последний совет насчет атрибутов экземпляров следует из того, что по умолчанию Python хранит атрибуты экземпляров в специальном атрибуте `__dict__`, который представляет собой словарь, являющийся принадлежностью эк-

[1] Именно так данные хранятся в кортежах.

экземпляра[1]. С тех пор как в версии Python 3.3 было реализовано предложение PEP 412 «Key-Sharing Dictionary» (*https://peps.python.org/pep-0412/*), экземпляры класса могут разделять общую хеш-таблицу, хранящуюся в самом классе. На эту общую хеш-таблицу ссылаются атрибуты `__dict__` каждого нового экземпляра, имеющего такие же имена атрибутов, что и первый экземпляр этого класса, возвращенный `__init__`. Тогда `__dict__` в каждом экземпляре может хранить только собственные значения атрибутов в виде простого массива указателей. Добавление атрибута экземпляра после возврата из `__init__` заставляет Python создать новую хеш-таблицу для хранения `__dict__` только одного этого экземпляра (такое поведение подразумевалось по умолчанию для всех экземпляров до версии Python 3.3). Согласно PEP 412, эта оптимизация сокращает потребление памяти в объектно-ориентированных программах на 10–20 %.

Детали компактного размещения в памяти и оптимизаций, связанных с разделением ключей, довольно сложны. Дополнительные сведения можно найти в статье «Internals of sets and dicts» (*https://www.fluentpython.com/extra/internals-of-sets-and-dicts/*) на сайте *fluentpython.com*.

Теперь перейдем к множествам.

Теория множеств

Множества – сравнительно недавнее добавление к Python, которое используется недостаточно широко. Тип `set` и его неизменяемый вариант `frozenset` впервые появились в виде модуля в Python 2.3, а в Python 2.6 были «повышены» до встроенных типов.

В этой книге словом «множество» обозначается как `set`, так и `frozenset`.

Множество – это набор уникальных объектов. Поэтому один из основных способов его использования – устранение дубликатов:

```
>>> l = ['spam', 'spam', 'eggs', 'spam', 'bacon', 'eggs']
>>> set(l)
{'eggs', 'spam', 'bacon'}
>>> list(set(l))
['eggs', 'spam', 'bacon']
```

Если вы хотите устранить дубликаты, сохранив при этом порядок первого вхождения каждого элемента, можете воспользоваться простым словарем `dict`:

```
>>> dict.fromkeys(l).keys()
dict_keys(['spam', 'eggs', 'bacon'])
>>> list(dict.fromkeys(l).keys())
['spam', 'eggs', 'bacon']
```

[1] Если только в классе нет атрибута `__slots__`, назначение которого объясняется в разделе «Экономия памяти с помощью __slots__» ниже.

Элементы множества должны быть хешируемыми. Сам тип `set` хешируемым не является, поэтому объекты `set` не могут вложенными. Но тип `frozenset` хешируемый, поэтому элементами `set` могут быть объекты типа `frozenset`.

Помимо гарантии уникальности, типы множества предоставляют набор теоретико-множественных операций, в частности инфиксные операции: если `a` и `b` – множества, то `a | b` – их объединение, `a & b` – пересечение, а `a - b` – разность. Умелое пользование теоретико-множественными операциями помогает уменьшить как объем, так и время работы Python-программ и одновременно сделать код более удобным для восприятия и осмысления – за счет устранения циклов и условных конструкций.

Пусть, например, у нас есть большой набор почтовых адресов (`haystack`, стог) и меньший набор адресов (`needles`, иголок), а наша задача – подсчитать, сколько раз элементы `needles` встречаются в `haystack`. Благодаря операции пересечения множеств (оператор `&`) для ее решения достаточно одной строки (пример 3.12).

Пример 3.12. Подсчет количества вхождений `needles` в `haystack`, оба объекта имеют тип `set`

```
found = len(needles & haystack)
```

Без оператора пересечения эту программу пришлось бы написать, как показано в примере 3.13.

Пример 3.13. Подсчет количества вхождений `needles` в `haystack` (результат тот же, что в примере 3.10)

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

Программа из примера 3.12 работает чуть быстрее, чем из примера 3.13. С другой стороны, пример 3.13 работает для любых итерируемых объектов `needles` и `haystack`, тогда как в примере 3.12 требуется, чтобы оба были множествами. Впрочем, если исходные объекты множествами не были, то их легко можно построить на лету, как показано в примере 3.14.

Пример 3.14. Подсчет количества вхождений `needles` в `haystack`; этот код работает для любых итерируемых типов

```
found = len(set(needles) & set(haystack))

# или по-другому:
found = len(set(needles).intersection(haystack))
```

Разумеется, построение множеств в примере 3.14 обходится не бесплатно, но если `needles` или `haystack` уже является множеством, то варианты, показанные в примере 3.14, могут оказаться дешевле кода из примера 3.13.

Любой из показанных выше примеров тратит на поиск 1000 «иголок» в «стоге» `haystack`, состоящем из 10 000 000 элементов, чуть больше 0.3 миллисекунды, т. е. по 0.3 микросекунды на одну «иголку».

Помимо чрезвычайно быстрой проверки вхождения (благодаря механизму хеш-таблиц), встроенные типы `set` и `frozenset` предоставляют богатый набор

операций для создания новых множеств или – в случае `set` – модификации существующих. Ниже мы обсудим эти операции, но сначала сделаем одно замечание о синтаксисе.

Литеральные множества

Синтаксис литералов типа `set` – `{1}`, `{1, 2}` и т. д. – выглядит в точности как математическая нотация, за одним важным исключением: не существует литерального обозначения пустого множества, в таком случае приходится писать `set()`.

Синтаксический подвох

Не забывайте: для создания пустого множества `set` следует использовать конструктор без аргументов: `set()`. Написав `{}`, вы, как и в прошлых версиях, создадите пустой словарь `dict`.

В Python 3 для представления множеств строками используется нотация `{...}` во всех случаях, кроме пустого множества:

```
>>> s = {1}
>>> type(s)
<class 'set'>
>>> s
{1}
>>> s.pop()
1 >>> s
set()
```

Литеральный синтаксис множеств вида `{1, 2, 3}` быстрее и понятнее, чем вызов конструктора (например, `set([1, 2, 3])`). При этом вторая форма медленнее, потому что для вычисления такого выражения Python должен найти класс `set` по имени, чтобы получить его конструктор, затем построить список и, наконец, передать этот список конструктору. А при обработке литерала `{1, 2, 3}` Python исполняет специализированный байт-код `BUILD_SET`[1].

Не существует специального синтаксиса для литералов, представляющих `frozenset`, – их приходится создавать с помощью конструктора. И стандартное строковое представление в Python 3 выглядит как вызов конструктора `frozenset`. Ниже показан пример в сеансе оболочки:

```
>>> frozenset(range(10))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

И раз уж мы заговорили о синтаксисе, то отметим, что хорошо знакомый синтаксис спискового включения был приспособлен и для построения множеств.

---

[1] Это, возможно, интересно, но не так уж важно. Ускорение достигается, только когда литеральное множество вычисляется, а это происходит не более одного раза в каждом процессе Python – при первоначальной компиляции модуля. Особо любознательные могут импортировать функцию `dis` из модуля `dis` и с ее помощью дизассемблировать байт-код создания литерального множества, например `dis('{1}')`, и вызова конструктора `set` – `dis('set([1])')`.

Множественное включение

Множественное включение (*setcomp*) было добавлено еще в версии Python 2.7 наряду со словарным включением, рассмотренным выше. См. пример 3.15.

Пример 3.15. Построение множества символов Latin-1, в Unicode-названии которых встречается слово «SIGN»

```
>>> from unicodedata import name ❶
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i),'')} ❷
{'§', '=', '¢', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', '©',
'°', '+', '÷', '±', '>', '¬', '®', '%'}
```

❶ Импортировать функцию `name` из `unicodedata` для получения названий символов.

❷ Построить множество символов с кодами от 32 до 255, в названиях которых встречается слово «SIGN»

Порядок вывода символов будет разным в разных процессах Python из-за случайного начального значения при создании хеша (см. раздел «Что такое “хешируемый”?».

Но оставим в стороне вопросы синтаксиса и перейдем к поведению множеств.

Практические последствия внутреннего устройства класса set

Типы `set` и `frozenset` реализованы с помощью хеш-таблиц. Отсюда вытекает ряд следствий.

- Элементы множества должны быть хешируемыми объектами. Ключи должны быть хешируемыми объектами. Они должны правильно реализовывать методы `__hash__` и `__eq__`, как описано в разделе «Что значит “хешируемый”?».
- Проверка на членство производится очень эффективно. Множество может содержать миллионы элементов, но для нахождения элемента нужно только вычислить его хеш-код и получить по нему смещение относительно начала хеш-таблицы; возможно, еще придется выполнить несколько попыток, чтобы добраться до нужного элемента или убедиться, что его не существует.
- Множествам присущи большие накладные расходы на хранение в памяти по сравнению с низкоуровневым массивом указателей на элементы, что было бы компактнее, но резко замедлило бы поиск, если число элементов в множестве сколько-нибудь велико.
- Порядок элементов зависит от порядка вставки, но опираться на эту зависимость не рекомендуется, т. к. это ненадежно. Если два элемента различны, но имеют одинаковый хеш-код, то их взаимное расположение зависит от того, какой элемент был добавлен первым.
- Добавление элементов в множество может изменить порядок уже присутствующих в нем элементов. Это связано с тем, что эффективность алгоритма падает, когда хеш-таблица заполнена на две трети или более,

поэтому Python приходится увеличивать размер таблицы по мере ее роста, что влечет за собой перемещение в памяти. Когда такое происходит, элементы вставляются заново, и их относительный порядок может измениться.

Дополнительные сведения можно найти в статье «Internals of sets and dicts» (*https://www.fluentpython.com/extra/internals-of-sets-and-dicts/*) на сайте *fluentpython.com*.

Теперь рассмотрим богатый ассортимент операций над множествами.

Операции над множествами

На рис. 3.2 приведена сводка методов, которые имеются у изменяемых и неизменяемых множеств. Многие из них – специальные методы, поддерживающие перегрузку операторов. В табл. 3.2 показаны математические операции над множествами, которым соответствуют какие-то операторы или методы в Python. Отметим, что некоторые операторы и методы изменяют конечное множество на месте (например, `&=`, `difference_update` и т. д.). Таким операциям нет места в идеальном мире математических множеств, и в классе `frozenset` они не реализованы.

Инфиксные операторы, приведенные в табл. 3.2, требуют, чтобы оба операнда были множествами, но все остальные методы принимают в качестве аргументов один или несколько итерируемых объектов. Например, чтобы создать объединение четырех коллекций `a`, `b`, `c`, `d`, можно написать `a.union(b, c, d)`, где `a` должно иметь тип `set` и `b`, `c` и `d` могут быть итерируемыми объектами любого типа, порождающего хешируемые элементы. Если требуется создать новое множество, являющееся объединением четырех итерируемых объектов, то вместо модификации существующего множества можно написать `{*a, *b, *c, *d}`. Это предложено в документе PEP 448 «Additional Unpacking Generalizations» (*https://peps.python.org/pep-0448/*) и реализовано начиная с версии Python 3.5.

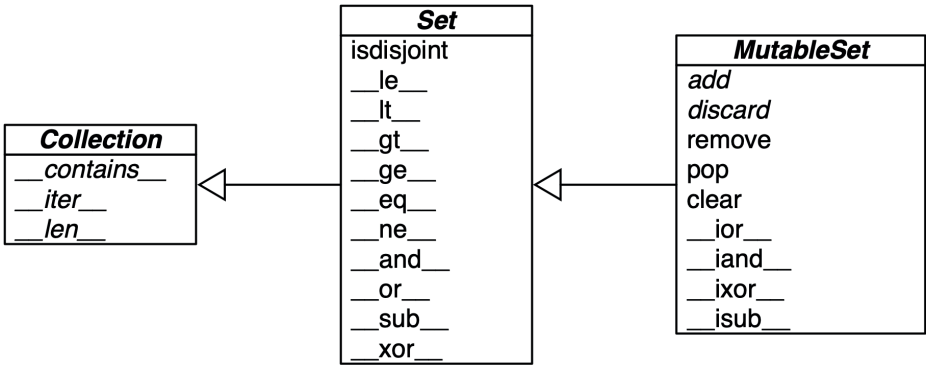


Рис. 3.2. UML-диаграмма класса `MutableSet` и его суперклассов из модуля `collections.abc` (курсивом набраны имена абстрактных классов и абстрактных методов, инверсные операторные методы для краткости опущены)

Таблица 3.2. Математические операции над множествами: эти методы либо порождают новое множество, либо модифицируют конечное множество на месте (если оно изменяемое)

Мат. символ	Оператор Python	Метод	Описание
S ∩ Z	`s & z`	`s.__and__(z)`	Пересечение `s` и `z`
		`s.intersection(it,…)`	Пересечение `s` и всех множеств, построенных из итерируемых объектов `it` и т. д.
	`s &= z`	`s.__iand__(z)`	Замена `s` пересечением `s` и `z`
		`s.intersection_update(it,…)`	Замена `s` пересечением `s` и всех множеств, построенных из итерируемых объектов `it` и т. д.
S ∪ Z	`s | z`	`s.__or__(z)`	Объединение `s` и `z`
	`z | s`	`z.__ror__(s)`	Инверсный оператор `|`
		`s.union(it,…)`	Объединение `s` и всех множеств, построенных из итерируемых объектов `it` и т. д.
	`s |= z`	`s.__ior__(z)`	Замена `s` объединением `s` и `z`
		`s.update(it,…)`	Замена `s` объединением `s` и всех множеств, построенных из итерируемых объектов `it` и т. д.
S \ Z	`s - z`	`s.__sub__(z)`	Относительное дополнение или разность `s` и `z`
	`z - s`	`z.__rsub__(s)`	Инверсный оператор `-`
		`s.difference(it,…)`	Разность между `s` и всеми множествами, построенными из итерируемых объектов `it` и т. д.
	`s -= z`	`s.__isub__(z)`	Замена `s` разностью между `s` и `z`
		`s.difference_update(it,…)`	Замена `s` разностью между `s` и всеми множествами, построенными из итерируемых объектов `it` и т. д.
		`s.symmetric_difference(it)`	Дополнение `s & set(it)`
S Δ Z	`s ^ z`	`s.__xor__(z)`	Симметрическая разность (дополнение пересечения `s & z`)
	`z ^ s`	`z.__rxor__(s)`	Инверсный оператор `^`
		`s.symmetric_difference_update(it,…)`	Замена `s` симметрической разностью между `s` и всеми множествами, построенными из итерируемых объектов `it` и т. д.
	`s ^= z`	`s.__ixor__(z)`	Замена `s` симметрической разностью между `s` и `z`

В табл. 3.3 перечислены теоретико-множественные предикаты: методы и операторы, которые возвращают `True` или `False`.

Таблица 3.3. Операторы сравнения множеств и методы, возвращающие булево значение

Мат. символ	Оператор Python	Метод	Описание
`S ∩ Z = ∅`		`s.isdisjoint(z)`	`s` и `z` дизъюнктны (т. е. не пересекаются)
`e ∈ Z`	`e in s`	`s.__contains__(e)`	`e` является элементом `s`
`S ⊆ Z`	`s <= z`	`s.__le__(z)`	`s` является подмножеством `z`
		`s.issubset(it)`	`s` является подмножеством множества `z`, построенного из итерируемого объекта `it`
`S ⊂ Z`	`s < z`	`s.__lt__(z)`	`s` является собственным подмножеством `z`
`S ⊇ Z`	`s >= z`	`s.__ge__(z)`	`s` является надмножеством `z`
		`s.issuperset(it)`	`s` является надмножеством множества `z`, построенного из итерируемого объекта `it`
`S ⊃ Z`	`s > z`	`s.__gt__(z)`	`s` является собственным надмножеством `z`

Помимо теоретико-множественных операторов и методов, типы множеств реализуют и другие методы, полезные на практике. Они сведены в табл. 3.4.

Таблица 3.4. Дополнительные методы множеств

	set	frozenset	
`s.add(e)`	●		Добавить элемент `e` в `s`
`s.clear()`	●		Удалить все элементы из `s`
`s.copy()`	●	●	Поверхностная копия `s`
`s.discard(e)`	●		Удалить элемент `e` из `s`, если он там присутствует
`s.__iter__()`	●	●	Получить итератор для обхода `s`
`s.__len__()`	●	●	`len(s)`
`s.pop()`	●		Удалить и вернуть элемент `s`, возбудив исключение `KeyError`, если `s` пусто
`s.remove(e)`	●		Удалить элемент `e` из `s`, возбудив исключение `KeyError`, если `e` отсутствует в `s`

На этом мы завершаем обзор множеств и их возможностей. Теперь, как и было обещано в разделе «Представления словарей», убедимся, что два типа представления словарей ведут себя очень похоже на `frozenset`.

Теоретико-множественные операции над представлениями словарей

В табл. 3.5 показано, что объекты представлений, возвращаемые методами `.keys()` и `.items()` объекта `dic`, удивительно похожи на `frozenset`.

Таблица 3.4. Дополнительные методы множеств

	frozenset	dict_keys	dict_items	Описание
`s.__and__(z)`	●	●	●	`s & z` (пересечение `s` и `z`)
`s.__rand__(z)`	●	●	●	Инверсный оператор &
`s.__contains__()`	●	●	●	`e` является элементом `s`
`s.copy()`	●			Поверхностная копия `s`
`s.difference(it,…)`	●			Разность между `s` и итерируемыми объектами `it` и т. д.
`s.intersection(it,…)`	●			Пересечение `s` и итерируемых объектов `it` и т. д.
`s.isdisjoint(z)`	●	●	●	`s` и `z` дизъюнктны (т. е. не имеют общих элементов)
`s.issubset(it)`	●			`s` является подмножеством итерируемого объекта `it`
`s.issuperset(it)`	●			`s` является надмножеством итерируемого объекта `it`
`s.__iter__()`	●	●	●	Получить итератор для обхода `s`
`s.__len__()`	●	●	●	`len(s)`
`s.__or__(z)`				`s | z` (объединение `s` и `z`)
`s.__ror__(z)`	●	●	●	Инверсный оператор |
`s.__reversed__()`		●	●	Получить итератор для обхода `s` в обратном порядке
`s.__rsub__(z)`	●	●	●	Инверсный оператор -
`s.__sub__(z)`	●	●	●	`s - z` (разность `s` и `z`)
`s.symmetric_difference(it)`	●			Дополнение `s & set(it)`
`s.union(it,…)`	●			Объединение `s` и итерируемых объектов `it` и т. д.
`s.__xor__(z)`	●	●	●	`s ^ z` (симметрическая разность `s & z`)
`s.__rxor__(z)`	●	●	●	Инверсный оператор ^

В частности, `dict_keys` и `dict_items` реализуют специальные методы для поддержки операторов над множествами `&` (пересечение), `|` (объединение), `-` (разность) и `^` (симметрическая разность).

Например, с помощью `&` легко получить ключи, встречающиеся в двух словарях:

```
>>> d1 = dict(a=1, b=2, c=3, d=4)
>>> d2 = dict(b=20, d=40, e=50)
>>> d1.keys() & d2.keys()
{'b', 'd'}
```

Отметим, что оператор `&` возвращает значение типа `set`. Более того, операторы над множествами в представлениях словарей совместимы с экземплярами `set`. Убедимся в этом:

```
>>> s = {'a', 'e', 'i'}
>>> d1.keys() & s
{'a'}
>>> d1.keys() | s
{'a', 'c', 'b', 'd', 'i', 'e'}
```

Представление `dict_items` работает как множество, только если все значения в словаре допускают хеширование. Попытка применить операцию над множествами к представлению `dict_items` с нехешируемыми значениями приводит к исключению `TypeError: unhashable type 'T'`, где `T` – тип недопустимого значения.

С другой стороны, представление `dict_keys` всегда можно использовать как множество, потому что все ключи словаря являются хешируемыми по определению.

Применение операторов множества к представлениям позволяет отказаться от многочисленных циклов и условных предложений при исследовании содержимого словарей. Заставьте эффективную реализацию Python, написанную на C, работать для вас!

На этом мы можем подвести итоги.

Резюме

Словари – краеугольный камень Python. С годами знакомый синтаксис литералов `{k1: v1, k2: v2}` пополнился поддержкой распаковки с помощью оператора `**`, сопоставлением с образцом и словарным включением.

Помимо базового класса `dict`, в стандартной библиотеке имеются удобные, готовые к применению специализированные отображения, например `defaultdict`, `OrderedDict`, `ChainMap` и `Counter`, все они определены в модуле `collections`. После внедрения новой реализации `dict` класс `OrderedDict` уже не так полезен, как прежде, но остается в стандартной библиотеке ради обратной совместимости; к тому же он обладает рядом свойств, отсутствующих у `dict`, в частности принимает во внимание порядок ключей в сравнениях словарей на равенство `==`. Также в модуле `collections` находится предназначенный для расширения класс `UserDict`.

Два весьма полезных метода, имеющихся в большинстве отображений, – `setdefault` и `update`. Метод `setdefault` используется для модификации элементов, содержащих изменяемые значения, например в словаре значений типа `list`, чтобы избежать повторных операций поиска того же ключа. Метод `update` облегчает массовую вставку или перезапись элементов, когда новые элементы берутся из другого отображения, из итерируемого объекта, порождающего пары `(key, value)`, или из именованных аргументов. Конструкторы отображений также пользуются методом `update`, что позволяет инициализировать отображение другим отображением, итерируемым объектом или именованными аргументами. Начиная с версии Python 3.9 мы также можем использовать оператор `|=` для модификации отображения и оператор `|` для создания нового отображения из объединения двух существующих.

В API отображений имеется метод `__missing__`, который позволяет определить, что должно происходить в случае отсутствия ключа при использовании синтаксической конструкции `d[k]`, которая вызывает метод `__getitem__`.

Модуль `collections.abc` содержит абстрактные базовые классы `Mapping` и `MutableMapping` для справки и контроля типов. Класс `MappingProxyType` из модуля `types` позволяет создавать неизменяемый фасад для отображения, которое требуется защитить от непреднамеренного изменения. Существуют также абстрактные базовые классы `Set` и `MutableSet`.

Представления словарей стали замечательным приобретением в Python 3, поскольку позволили исключить дополнительный расход памяти при использовании методов `.keys()`, `.values()` и `.items()`, которые в Python 2 строили списки, дублирующие данные, уже присутствовавшие в экземпляре `dict`. Кроме того, классы `dict_keys` и `dict_items` поддерживают наиболее полезные операторы и методы класса `frozenset`.

Дополнительная литература

В разделе документации по стандартной библиотеке Python «collections – контейнерные типы данных» (*https://docs.python.org/3/library/collections.html*) есть примеры и практические рецепты использования различных типов отображений. Исходный код модуля *Lib/collections/init.py* станет отличным справочным пособием для всех, кто захочет написать новый тип отображения или разобраться в логике работы существующих. В главе 1 книги David Beazley, Brian K. Jones «*Python Cookbook*», 3-е издание (O'Reilly), имеется 20 полезных и поучительных примеров работы со структурами данных – в большинстве из них изобретательно используется словарь `dict`.

Грег Гандербергер выступает за то, чтобы и дальше использовать класс `collections.OrderedDict` на том основании, что «явное лучше неявного», а также ради обратной совместимости и потому что некоторые инструменты и библиотеки предполагают, что порядок ключей в `dict` несущественен. См. его статью «Python Dictionaries Are Now Ordered. Keep Using OrderedDict» (*http://gandenberger.org/2018/03/10/ordered-dicts-vs-ordereddict/*).

Документ PEP 3106 «Revamping dict.keys(), .values() and .items()» (*https://peps.python.org/pep-3106/*) – то место, где Гвидо ван Россум впервые описал представления словарей для Python 3. В реферате он написал, что идея навеяна коллекциями в Java.

PyPy (*https://www.pypy.org/*) стал первым интерпретатором Python, в котором было реализовано предложение Раймонда Хэттингера по поводу компактных словарей. Авторы рассказали об этом в статье «Faster, more memory efficient and more ordered dictionaries on PyPy» (*https://morepypy.blogspot.com/2015/01/faster-more-memory-efficient-and-more.html*), признав, что похожее размещение в памяти использовалось в PHP 7 и было описано в статье «PHP's new hashtable implementation» (*https://www.npopov.com/2014/12/22/PHPs-new-hashtable-implementation.html*). Всегда приятно, когда авторы ссылаются на достижения своих предшественников.

На конференции PyCon 2017 Брэндон Родес провел презентацию «The Dictionary Even Mightier» (*https://www.youtube.com/watch?v=66P5FMkWoVU*),

продолжение его классической анимированной презентации «The Mighty Dictionary» (*https://pyvideo.org/pycon-us-2010/the-mighty-dictionary-55.html*), включавшей анимированные коллизии в хеше! Еще одно современное, но более глубокое видео, посвященное внутреннему устройству словарей в Python, – «Modern Dictionaries» (*https://www.youtube.com/watch?v=p33CVV29OG8*) Раймонда Хэттингера, где он рассказывает, что после первой неудачной попытки «впарить» компактные словари разработчикам ядра CPython он обратился к команде PyPy, те приняли его идею, она вызвала интерес, и в конечном итоге Инада Наоки включил ее в CPython 3.6 (*https://docs.python.org/3/whatsnew/3.6.html#new-dict-implementation*). Полную информацию можно почерпнуть из пространного комментария к файлу *Objects/dictobject.c* (*https://github.com/python/cpython/blob/cf7eaa4617295747ee5646 c4e2b7e7a16d7c64ab/Objects/dictobject.c*) в коде CPython и в техническом документе *Objects/dictnotes.txt* (*https://github.com/python/cpython/blob/cf7eaa4617295747ee5646c 4e2b7e7a16d7c64ab/Objects/dictnotes.txt*).

Аргументация в пользу добавления множеств в язык приведена в документе PEP 218 «Adding a Built-In Set Object Type» (*https://www.python.org/dev/peps/pep-0218/*). Когда этот документ был одобрен, для множеств еще не была принята какая-то специальная литеральная нотация. Литеральные множества появились в Python 3, а затем были внедрены и в версию Python 2.7 наряду со словарными и множественными включениями. На конференции PyCon 2019 я провел презентацию «Set Practice: learning from Python's set types» (*https://speakerdeck.com/ramalho/python-set-practice-at-pycon*), в которой описал случаи использования множеств в реальных программах, рассмотрел структуру их API и реализацию `uintset` (*https://github.com/ramalho/uintset*) – класса множества целых чисел с использованием битового вектора вместо хеш-таблицы, идея которого была навеяна примером из главы 6 замечательной книги Alan Donovan and Brian Kernighan «The Go Programming Language» (издательство Addison-Wesley)[1].

В журнале *Spectrum* (), издаваемом IEEE, напечатан рассказ о Хансе Петере Луне, плодовитом изобретателе, который запатентовал колоду перфокарт для выбора рецептов коктейлей в зависимости от доступных ингредиентов, а также ряд других изобретений, в том числе … хеш-таблицы! См. статью «Hans Peter Luhn and the Birth of the Hashing Algorithm» (*https://spectrum.ieee.org/hans-peter-luhn-and-the-birth-of-the-hashing-algorithm*).

Поговорим

Синтаксический сахар

Мой друг Джеральдо Коэн как-то заметил, что Python – «простой и корректный язык».

Ревнители чистоты языков программирования часто отметают синтаксис как нечто несущественное.

> Синтаксический сахар вызывает рак точек с запятой.
>
> – Алан Перлис

[1] *Алан Донован, Брайан У. Керниган.* Язык программирования Go. Диалектика-Вильямс, 2020.

Синтаксис – это пользовательский интерфейс языка программирования, поэтому на практике он очень важен.

До того как я открыл для себя Python, я писал веб-приложения на Perl, PHP и JavaScript. Мне очень нравился литеральный синтаксис отображений в этих языках, которого так не хватало в Java и C.

Хороший синтаксис литеральных отображений упрощает конфигурирование, реализации на основе таблиц и хранение данных для создания прототипов и тестирования. Это тот урок, который извлекли проектировщики Go из анализа динамических языков. Отсутствие такого синтаксиса для выражения структурированных данных в самом коде вынудило сообщество Java принять многословный и чрезмерно сложный язык XML в качестве формата данных.

Формат JSON был предложен как «обезжиренная альтернатива XML» (*http://www.json.org/fatfree.html*) и добился ошеломительного успеха, заменив XML во многих контекстах. Благодаря краткому синтаксису списков и словарей он отлично подходит на роль формата для обмена данными.

PHP и Ruby позаимствовали синтаксис хеша из Perl, где для ассоциации ключей и значений применяется оператор `=>`. JavaScript последовал по стопам Python и использует для этой цели знак `:`. Зачем использовать два символа, когда для удобочитаемости хватает и одного?

Истоки JSON следует искать в JavaScript, но так получилось, что это почти точное подмножество синтаксиса Python. JSON совместим с Python во всем, кроме написания значений `true`, `false` и `null`.

Армин Ронахер в своем твите (*https://fpy.li/3-33*) пишет, что ему понравилось «подправить» глобальное пространство имен Python, добавив совместимые с JSON псевдонимы для символов Python `True`, `False` и `None`, так что теперь он может копировать JSON прямо на консоль. Вот как выглядит его идея:

```
>>> true, false, null = True, False, None
>>> fruit = {
... "type": "banana",
... "avg_weight": 123.2,
... "edible_peel": false,
... "species": ["acuminata", "balbisiana", "paradisiaca"],
... "issues": null,
... }
>>> fruit
{'type': 'banana', 'avg_weight': 123.2, 'edible_peel': False,
'species': ['acuminata', 'balbisiana', 'paradisiaca'], 'issues':
  None}
```

Синтаксис, которым ныне все пользуются для обмена данными, – это синтаксис словаря и списка в Python. Теперь у нас есть изящный синтаксис, который к тому же сохраняет порядок вставки.

Просто и корректно.

Глава 4

Unicode-текст и байты

> Человек работает с текстом, компьютер – с байтами.
>
> – Эстер Нэм и Трэвис Фишер, «Кодировка символов и Unicode в Python»[1]

В Python 3 появилось четкое различие между строками текста, предназначенными для человека, и последовательностями байтов. Неявное преобразование последовательности байтов в Unicode-текст ушло в прошлое. В этой главе речь пойдет о Unicode-строках, двоичных последовательностях и кодировках для преобразования одного в другое.

Так ли важно глубоко разбираться в Unicode? Ответ на этот вопрос зависит от того, в какой области вы программируете на Python. Но, так или иначе, от различий между типами `str` и `byte` никуда не деться. А в качестве премии за потраченные усилия вы узнаете, что специализированные типы двоичных последовательностей обладают возможностями, которых нет у «универсального» типа `str` в Python 2.

В этой главе мы рассмотрим следующие вопросы:

- символы, кодовые позиции и байтовые представления;
- уникальные особенности двоичных последовательностей: `bytes`, `bytearray` и `memoryview`;
- кодировки для полного Unicode и унаследованных наборов символов;
- как предотвращать и обрабатывать ошибки кодировки;
- рекомендации по работе с текстовыми файлами;
- кодировка по умолчанию и стандартные проблемы ввода-вывода;
- безопасное сравнение Unicode-текстов с нормализацией;
- служебные функции для нормализации, сворачивания регистра и явного удаления диакритических знаков;
- правильная сортировка Unicode-текстов с помощью модуля `locale` и библиотеки *pyuca*;
- символьные метаданные в базе данных Unicode;
- двухрежимные API для работы с типами `str` и `bytes`.

[1] Слайд 12 выступления на конференции PyCon 2014 «Character Encoding and Unicode in Python» (слайды – *https://www.slideshare.net/fischertrav/character-encoding-unicode-how-to-with-dignity-33352863*, видео – *https://pyvideo.org/pycon-us-2014/character-encoding-and-unicode-in-python.html*).

Что нового в этой главе

Поддержка Unicode в Python 3 достаточно полная и стабильная, поэтому самое заметное добавление – раздел «Поиск символов по имени», в котором описана утилита для поиска в базе данных Unicode – отличный способ искать цифры в кружочках и улыбающихся котят прямо из командной строки.

Заслуживает также упоминания небольшое изменение в поддержке Unicode в Windows, которая стала лучше и проще начиная с версии Python 3.6, в чем у нас будет случай убедиться в разделе «Остерегайтесь кодировок по умолчанию».

Начнем с не особенно новых, но фундаментальных понятий: символов, кодовых позиций и байтов.

Для второго издания я расширил раздел, посвященный модулю `struct`, и опубликовал его в виде статьи «Parsing binary records with struct» (*https://www.fluentpython.com/extra/parsing-binary-struct/*) на сопроводительном сайте *fluentpython.com*.

Там же вы найдете статью «Building Multi-character Emojis» (*https://www.fluentpython.com/extra/multi-character-emojis/*), в которой описано, как создавать эмодзи с изображением флагов стран, людей с разным цветом кожи и значки из различных семейств путем комбинирования символов Unicode.

О символах, и не только

Концепция «строки» достаточно проста: строка – это последовательность символов. Проблема – в определении понятия «символ».

В 2021 году под «символом» мы понимаем символ Unicode, и это лучшее определение на сегодняшний момент. Поэтому отдельными элементами объекта типа `str` в Python 3 являются символы Unicode (точно так же обстоит дело с элементами объекта `unicode` в Python 2), – а не просто байты, из которых состоят объекты `str` в Python 2.

Стандарт Unicode явно разделяет идентификатор символа и конкретное байтовое представление.

Идентификатор символа – его *кодовая позиция* – это число от 0 до 1 114 111 (по основанию 10), которое в стандарте Unicode записывается шестнадцатеричными цифрами (в количестве от 4 до 6) с префиксом «U+». Например, кодовая позиция буквы A равна U+0041, знака евро – U+20AC, музыкального символа скрипичного ключа – U+1D11E. В версии Unicode 13.0.0 (используемой в Python 3.10.0b4) конкретные символы сопоставлены примерно 13 % допустимых кодовых позиций.

Какими именно байтами представляется символ, зависит от используемой *кодировки*. Кодировкой называется алгоритм преобразования кодовых позиций в последовательности байтов и наоборот. Кодовая позиция буквы A (U+0041) кодируется одним байтом `\x41` в кодировке UTF-8 и двумя байтами `\x41\x00` в кодировке UTF-16LE. Другой пример: знак евро (U+20AC) преобразуется в три байта в UTF-8 – `\xe2\x82\xac`, но в UTF-16LE кодируется двумя байтами – `\xac\x20`.

Преобразование из кодовых позиций в байты называется *кодированием*, преобразование из байтов в кодовые позиции – *декодированием*. См. пример 4.1.

Пример 4.1. Кодирование и декодирование

```
>>> s = 'café'
>>> len(s) ❶
4 >>> b
=
s.encode('utf8') ❷
>>> b
b'caf\xc3\xa9' ❸
>>> len(b) ❹
5 >>> b.decode('utf8') ❺
'café'
```

❶ Строка `'café'` состоит из четырех символов Unicode.
❷ Преобразуем `str` в `bytes`, пользуясь кодировкой UTF-8.
❸ Литералы типа `bytes` начинаются префиксом `b`.
❹ Объект `b` типа `bytes` состоит из пяти байт (кодовая позиция, соответствующая «é», в UTF-8 кодируется двумя байтами).
❺ Преобразуем `bytes` обратно в `str`, пользуясь кодировкой UTF-8.

Если вы никак не можете запомнить, когда употреблять `.decode()`, а когда – `.encode()`, представьте, что последовательности байтов – это загадочный дамп памяти машины, а объекты Unicode `str` – «человеческий» текст. Тогда *декодирование* `bytes` в `str` призвано получить понятный человеку текст, а *кодирование* `str` в `bytes` – получить представление, пригодное для хранения или передачи.

Тип `str` в Python 3 – это, по существу, не что иное, как переименованный тип `unicode` из Python 2. Но вот тип `bytes` в Python 3 – не просто старый тип `str`, и с ним тесно связан тип `bytearray`. Поэтому имеет смысл сначала разобраться с типами двоичных последовательностей, а уж потом переходить к вопросам кодирования и декодирования.

Все, что нужно знать о байтах

Новые типы двоичных последовательностей во многих отношениях похожи на тип `str` в Python 2. Главное, что нужно знать, – это то, что существуют два основных встроенных типа двоичных последовательностей: неизменяемый тип `bytes`, появившийся в Python 3, и изменяемый тип `bytearray`, добавленный в Python 2.6[1]. В документации по Python иногда используется общий термин «байтовая строка» (byte string) для обозначения и `bytes`, и `bytearray`. Я стараюсь не употреблять этот создающий путаницу термин.

Каждый элемент `bytes` или `bytearray` – целое число от 0 до 255, а не односимвольная строка, как в типе `str` в Python 2 `str`. Однако срез двоичной последовательности всегда является двоичной последовательностью того же типа, даже если это срез длины 1. См. пример 4.2.

[1] В Python 2.6 и 2.7 был также тип `bytes`, но теперь это просто псевдоним типа `str`.

Пример 4.2. Пятибайтовая последовательность в виде `bytes` и `bytearray`

```
>>> cafe = bytes('café', encoding='utf_8') ❶
>>> cafe
2
b'caf\xc3\xa9'
>>> cafe[0] ❷
99
>>> cafe[:1] ❸
b'c'
>>> cafe_arr = bytearray(cafe)
>>> cafe_arr ❹
bytearray(b'caf\xc3\xa9')
>>> cafe_arr[-1:] ❺
bytearray(b'\xa9')
```

❶ `bytes` можно получить из `str`, если известна кодировка.
❷ Каждый элемент – целое число в диапазоне `range(256)`.
❸ Срезы `bytes` также имеют тип `bytes`, даже если срез состоит из одного байта.
❹ Для типа `bytearray` не существует литерального синтаксиса: в оболочке объекты этого типа представляются в виде конструктора `bytearray()`, аргументом которого является литерал типа `bytes`.
❺ Срез `bytesarray` также имеет тип `bytesarray`.

Тот факт, что `my_bytes[0]` возвращает `int`, а `my_bytes[:1]` – последовательность объектов `bytes` длины 1, вызывает удивление только потому, что мы привыкли, что для типа Python `str` имеет место равенство `s[0] == s[:1]`. Для всех остальных типов последовательностей один элемент – не то же самое, что срез длины 1.

Хотя двоичные последовательности – на самом деле последовательности целых чисел, в их литеральной нотации отражен тот факт, что часто они включают ASCII-текст. Поэтому применяются различные способы отображения, зависящие от значения каждого байта.

- Для байтов с десятичными кодами от 32 до 126 – от пробела до ~ – выводится сам символ ASCII.
- Для байтов, соответствующих символам табуляции, новой строки, возврата каретки и \, выводятся управляющие последовательности `\t`, `\n`, `\r` и `\\`.
- Если в последовательности байтов встречаются оба ограничителя ' и «, то вся последовательность заключается в одиночные кавычки ', а все символы ' внутри нее экранируются, т. е. записываются в виде `\'`[1].
- Для всех остальных байтов выводится шестнадцатеричное представление (например, нулевой байт представляется последовательностью `\x00`).

Именно поэтому в примере 4.2 мы видим представление `b'caf\xc3\xa9'`: первые три байта `b'caf'` принадлежат диапазону символов ASCII с графическим начертанием, последний – нет.

[1] Любопытный факт: символ ASCII «одиночная кавычка», который в Python по умолчанию используется как ограничитель строки, в стандарте Unicode называется APOSTROPHE. А настоящие одиночные кавычки асимметричны: левая имеет код U+2018, а правая – U+2019.

Оба типа, `bytes` и `bytearray`, поддерживают все методы типа `str`, кроме тех, что относятся к форматированию (`format`, `format_map`), и еще нескольких, прямо зависящих от особенностей Unicode, в том числе `casefold`, `isdecimal`, `isidentifier`, `isnumeric`, `isprintable` и `encode`. Это означает, что при работе с двоичными последовательностями мы можем пользоваться знакомыми методами строк, например `endswith`, `replace`, `strip`, `translate`, `upper` и десятками других, только аргументы должны иметь тип `bytes`, а не `str`. К двоичным последовательностям применимы и функции для работы с регулярными выражениями из модуля `re`, если регулярное выражение откомпилировано из двоичной последовательности, а не из `str`. Начиная с версии Python 3.5 оператор `%` снова работает с двоичными последовательностями[1].

Для двоичных последовательностей существует метод класса, отсутствующий в типе `str`: `fromhex`, который строит последовательность, разбирая пары шестнадцатеричных цифр, которые могут быть разделены пробелами, хотя это и необязательно.

```
>>> bytes.fromhex('31 4B CE A9')
b'1K\xce\xa9'
```

Другие способы построения объектов `bytes` и `bytearray` связаны с вызовом различных конструкторов:

- с именованными аргументами `str` и `encoding`;
- с итерируемым объектом, порождающим элементы со значениями от 0 до 255;
- с объектом, который реализует протокол буфера (например, `bytes`, `bytearray`, `memoryview`, `array.array`), при этом байты копируются из исходного объекта во вновь созданную двоичную последовательность.

До версии Python 3.5 можно было также вызывать конструкторы `bytes` и `bytearray`, передавая одно целое число, чтобы создать двоичную последовательность такого размера, инициализированную нулевыми байтами. Эта сигнатура была объявлена нерекомендуемой в Python 3.5 и исключена в Python 3.6. См. документ PEP 467 «Minor API improvements for binary sequences» (*https://www.python.org/dev/peps/pep-0467/*).

Построение двоичной последовательности из буфероподобного объекта – это низкоуровневая операция, которая может потребовать приведения типов. См. пример 4.3.

Пример 4.3. Инициализация байтов данными, хранящимися в массиве

```
>>> import array
>>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
>>> octets = bytes(numbers) ❷
>>> octets
b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00' ❸
```

❶ Код типа `'h'` означает создание массива коротких целых (16-разрядных).

[1] Он не работал в версиях Python с 3.0 до 3.4, что причиняло много страданий разработчикам, имеющим дело с двоичными данными. Возврат к прошлому документирован в PEP 461 «Adding % formatting to bytes and bytearray» (*https://peps.python.org/pep-0461/*).

❷ В объекте `octets` хранится копия байтов, из которых составлены числа в массиве `numbers`.
❸ Это десять байт, представляющих пять коротких целых.

Создание объекта `bytes` или `bytearray` из буфероподобного источника всегда сопровождается копированием байтов. Напротив, объекты типа `memoryview` позволяют разным двоичным структурам данных использовать одну и ту же область памяти, как мы видели в разделе «Представления областей памяти».

После этого краткого введения в типы двоичных последовательностей в Python рассмотрим их преобразования в строки и обратно.

Базовые кодировщики и декодировщики

В дистрибутиве Python имеется свыше 100 *кодеков* (кодировщик-декодировщик) для преобразования текста в байты и обратно. У каждого кодека есть имя, например `'utf_8'`, а часто еще и синонимы, например `'utf8'`, `'utf-8'` и `'U8'`. Имя можно передать в качестве аргумента `encoding` таким функциям, как `open()`, `str.encode()`, `bytes.decode()` и т. д. В примере 4.4 показан один и тот же текст, закодированный как три разные последовательности байтов.

Пример 4.4. Строка «El Niño», закодированная тремя кодеками, дает совершенно разные последовательности байтов

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xf1o'
utf_8 b'El Ni\xc3\xb1o'
utf_16 b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

На рис. 4.1 показано, какие байты различные кодеки генерируют для некоторых символов: от буквы «A» до символа скрипичного ключа. Отметим, что последние три кодировки многобайтовые, переменной длины.

char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
ۿ	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
┌	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
𝄞	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

Рис. 4.1. Двенадцать символов, их кодовые позиции и байтовые представления (в 16-ричном виде) в семи разных кодировках (звездочка означает, что в данной кодировке этот символ непредставим)

Звездочки на рис. 4.1 ясно показывают, что некоторые кодировки, в частности ASCII и даже многобайтовая кодировка GB2312, не способны представить все символы Unicode. Однако кодировки семейства UTF спроектированы так, чтобы была возможность представить любую кодовую позицию Unicode.

Кодировки на рис. 4.1 образуют достаточно репрезентативную выборку.

`latin1`, *или* `iso8859_1`

Важна, потому что лежит в основе других кодировок, в частности `cp1252`, и самого Unicode (отметим, что значения байтов `latin1` повторяются в столбце `cp1252` и даже в самих кодовых позициях).

`cp1252`

Надмножество `latin1`, разработанное Microsoft. Добавлены некоторые полезные символы, например фигурные кавычки и знак евро €. В некоторых приложениях для Windows эта кодировка называется «ANSI», хотя никакой стандарт ANSI по этому поводу не принимался.

`cp437`

Оригинальный набор символов для IBM PC, содержащий символы псевдографики. Несовместим с кодировкой `latin1`, которая появилась позже.

`gb2312`

Унаследованный стандарт кодирования упрощенных китайских иероглифов, используемый в континентальном Китае; одна из нескольких широко распространенных многобайтовых кодировок для азиатских языков.

`utf-8`

Самая употребительная 8-разрядная кодировка в вебе. По состоянию на июль 2021 года в исследовании W³Techs «Usage statstics of character encodings for websites» (*https://w3techs.com/technologies/overview/character_encoding*) утверждается, что на 97 % сайтов используется кодировка UTF-8. В первом издании книги приводилась цифра на сентябрь 2014 года: 81.4 %.

`utf-16le`

Одна из форм 16-разрядной схемы кодирования UTF-16; все кодировки семейства UTF-16 поддерживают кодовые позиции с номерами, большими U+FFFF, с помощью управляющих последовательностей, называемых «суррогатными парами».

UTF-16 заменила первоначальную 16-разрядную кодировку в Unicode 1.0 – UCS-2 – еще в 1996 году. UCS-2 все еще развернута во многих системах, но поддерживает только кодовые позиции с номерами до U+FFFF. По состоянию на 2021 год более 57 % распределенных кодовых позиций имеют номера больше U+10000, сюда относятся и столь популярные эмодзи.

Завершив обзор распространенных кодировок, перейдем к проблемам, возникающим в процессе кодирования и декодирования.

Проблемы кодирования и декодирования

Существует общее исключение `UnicodeError`, но возникающая ошибка почти всегда более специфична: либо `UnicodeEncodeError` (в случае преобразования `str` в двоичную последовательность), либо `UnicodeDecodeError` (в случае чтения двоичной последовательности в `str`). При загрузке модулей Python может также возникать исключение `SyntaxError` в случае неожиданной кодировки исходного кода. В следующих разделах мы расскажем, как обрабатывать такие ошибки.

Первое, на что нужно обращать внимание, получив ошибку Unicode, – точный тип исключения. Это `UnicodeEncodeError`, `UnicodeDecodeError` или какая-то другая ошибка (например, `SyntaxError`), свидетельствующая об ошибке кодирования? Это главное, что нужно знать для решения проблемы.

Обработка UnicodeEncodeError

В большинстве кодеков, не входящих в семейство UTF, представлено только небольшое подмножество символов Unicode. Если в ходе преобразования текста в байты оказывается, что символ отсутствует в конечной кодировке, то возбуждается исключение `UnicodeEncodeError`, если только методу или функции кодировки не передан аргумент `errors`, обеспечивающий специальную обработку. Поведение обработчиков ошибок демонстрируется в примере 4.5.

Пример 4.5. Кодирование текста в байты: успешное завершение и обработка ошибок

```
>>> city = 'São Paulo'
>>> city.encode('utf_8') ❶
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1') ❷
b'S\xe3o Paulo'
>>> city.encode('cp437') ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/.../lib/python3.4/encodings/cp437.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in
position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore') ❹
b'So Paulo'
>>> city.encode('cp437', errors='replace') ❺
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace') ❻
b'S&#227;o Paulo'
```

❶ Кодировки `'utf_?'` справляются с любой строкой `str`.
❷ `'iso8859_1'` также работает для строки `'São Paulo'`.
❸ `'cp437'` не может закодировать букву `'ã'` («а» с тильдой). Обработчик ошибок по умолчанию – `'strict'` – возбуждает исключение `UnicodeEncodeError`.

❹ Обработчик `error='ignore'` молча пропускает некодируемые символы, обычно это не слишком удачная идея.

❺ Обработчик `error='replace'` заменяет некодируемые символы знаком `'?'`; данные теряются, но пользователь хотя бы знает, что какая-то часть информации утрачена.

❻ `'xmlcharrefreplace'` заменяет некодируемые символы XML-компонентом. Если вы не можете использовать UTF, а потеря данных недопустима, то это единственный вариант.

Механизм обработки ошибок в модуле `codecs` расширяемый. Можно зарегистрировать дополнительные значения аргумента `errors`, передав строку и функцию обработки ошибок функции `codecs.register_error`. См. документацию по `codecs.register_error` (*https://docs.python.org/3/library/codecs.html#codecs.register_error*).

ASCII – общее подмножество всех известных мне кодировок, поэтому любая кодировка должна работать, если текст состоит только из ASCII-символов. В Python 3.7 добавлен (*https://docs.python.org/3/library/ stdtypes.html#str.isascii*), новый булев метод `str.isascii()`, который проверяет, что Unicode-текст на 100 % состоит из ASCII-символов. Если это так, то его можно представить в виде байтов в любой кодировке, не опасаясь исключения `UnicodeEncodeError`.

Обработка UnicodeDecodeError

Не каждый байт содержит допустимый символ ASCII, и не каждая последовательность байтов является допустимой в кодировке UTF-8 или UTF-16. Если при декодировании двоичной последовательности встретится неожиданный байт, то возникнет исключение `UnicodeDecodeError`.

С другой стороны, многие унаследованные 8-разрядные кодировки, например `'cp1252'`, `'iso8859_1'` и `'koi8_r'`, могут декодировать произвольный поток байтов, в т. ч. случайный шум, без ошибок. Поэтому если ваша программа ошибется в предположении о том, какая 8-разрядная кодировка используется, то будет молча декодировать мусор.

В русской традиции «мусорные» символы называются «крокозябрами», а в англоязычной «гремлинами», или «mojibake» (文字化け – по-японски «трансформированный текст»).

В примере 4.6 показано, как неправильно выбранный кодек может порождать крокозябры или исключение `UnicodeDecodeError`.

Пример 4.6. Декодирование строки в байты: успешное завершение и обработка ошибок

```
>>> octets = b'Montr\xe9al' ❶
>>> octets.decode('cp1252') ❷
'Montréal'
>>> octets.decode('iso8859_7') ❸
'Montrıal'
>>> octets.decode('koi8_r') ❹
```

```
'MontrИal'
>>> octets.decode('utf_8') ❺
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace' ❻
'Montr�al'
```

❶ Эти байты являются символами строки «Montréal» в кодировке `latin1`; `'\xe9'` – байт, соответствующий букве «é».
❷ Декодирование с помощью кодировки `'cp1252'` (Windows 1252) работает, потому что она является собственным надмножеством `latin1`.
❸ Кодировка ISO-8859-7 предназначена для греческого языка, поэтому байт `'\xe9'` интерпретируется неправильно, но исключение не возбуждается.
❹ KOI8-R – кодировка для русского языка. Теперь `'\xe9'` интерпретируется как русская буква «И».
❺ Кодек `'utf_8'` обнаруживает октеты, не являющиеся допустимой последовательностью байтов в кодировке UTF-8, и возбуждает исключение `UnicodeDecodeError`.
❻ При использовании обработчика ошибок `'replace'` байт `\xe9` заменяется символом «�» (кодовая позиция U+FFFD), официальным ЗАМЕНЯЮЩИМ СИМВОЛОМ в Unicode, который служит для представления неизвестных символов.

Исключение SyntaxError при загрузке модулей с неожиданной кодировкой

UTF-8 – подразумеваемая по умолчанию кодировка исходного кода в Python 3 так же, как ASCII была кодировкой по умолчанию в Python 2. При попытке загрузить *py*-модуль, содержащий данные не в кодировке UTF-8 и не имеющий объявления кодировки, будет выдано сообщение вида:

```
SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line
  1, but no encoding declared; see http://python.org/dev/peps/pep-0263/
  for details
```

Поскольку в системах GNU/Linux и OS X практически повсеместно развернута кодировка UTF-8, такая ошибка наиболее вероятна при открытии *py*-файла, созданного в Windows в кодировке `cp1252`. Отметим, что она происходит даже в Python для Windows, потому что в Python 3 по умолчанию на всех платформах подразумевается кодировка UTF-8.

Чтобы исправить ошибку, добавьте в начало файла магический комментарий, как показано в примере 4.8.

Пример 4-7. ola.py: «Hello, World!» по-португальски

```
# coding: cp1252

print('Olá, Mundo!')
```

Теперь, когда исходный код на Python 3 не ограничивается одной лишь кодировкой ASCII и по умолчанию подразумевается замечательная UTF-8, самое правильное «лечение» для исходного кода в унаследованной кодировке типа `'cp1252'` – преобразовать в UTF-8 и не заморачиваться комментариями `coding`. Если ваш редактор не поддерживает UTF-8, пора его поменять.

Предположим, что имеется некий текстовый файл, все равно, исходный код или стихотворение, но вы не знаете, в какой кодировке он записан. Как определить истинную кодировку? Ответ см. в следующем разделе.

Как определить кодировку последовательности байтов

Как узнать, в какой кодировке записана последовательность байтов? Короткий ответ: никак. Кто-то должен вам сообщить.

В некоторых коммуникационных протоколах и файловых форматах, например HTTP и XML, предусмотрены заголовки, в которых явно указывается, как закодировано содержимое. Можно быть уверенным, что поток байтов представлен не в кодировке ASCII, если он содержит значения, большие 127, а сам способ построения UTF-8 и UTF-16 исключает определенные последовательности байтов.

Предложение Лео о том, как распознать кодировку UTF-8

(Следующие несколько абзацев составляют содержание комментария технического рецензента Леонардо Рохаэля к черновой редакции этой книги.)

Кодировка UTF-8 спроектирована так, что случайная последовательность байтов или даже неслучайная, но представленная в кодировке, отличной от UTF-8, почти наверняка не будет декодирована как мусор в UTF-8, а приведет к исключению `UnicodeDecodeError`.

Причина в том, что в управляющих последовательностях UTF-8 никогда не используются ASCII-символы и в эти последовательности встроены битовые паттерны, из-за которых очень трудно по ошибке принять случайные данные за корректный код в UTF-8.

Поэтому если вам удается декодировать несколько байтов с кодами, большими 127, как UTF-8, то с большой вероятностью это UTF-8 и есть.

Имея дело с бразильскими онлайновыми службами, некоторые из которых надстроены над унаследованными серверными продуктами, я, случалось, вынужден был реализовывать следующую стратегию декодирования: попытаться декодировать как UTF-8, а исключение `UnicodeDecodeError` рассматривать как указание на кодировку `cp1252`. Некрасиво, но эффективно.

Однако известно, что в естественных языках есть свои правила и ограничения. Поэтому есть допустить, что поток байтов – это *простой текст* на естественном языке, то его кодировку можно попытаться определить с помощью различных эвристических правил и статистики. Например, если часто встречается байт `b'\x00'`,

то это, скорее всего, 16- или 32-разрядная кодировка, но не 8-разрядная схема, потому что нулевые байты в открытом тексте – очевидная ошибка. Если нередко встречается последовательность `b'\x20\x00'`, то это, наверное, символ пробела (U+0020) в кодировке UTF-16LE, а не малоизвестный символ U+2000 `EN QUAD`.

Именно так и работает пакет Chardet – универсальный детектор кодировки символов (*https://pypi.python.org/pypi/chardet*), – который пытается распознать одну из 30 поддерживаемых кодировок. Chardet – написанная на Python библиотека, которую вы можете включить в свою программу, а кроме нее, пакет содержит также командную утилиту `chardetect`. Вот что она сообщает о файле с исходным кодом к данной главе:

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

Хотя в самих двоичных последовательностях закодированного текста обычно нет явных указаний на кодировку, в некоторых UTF-форматах в начале файла может находиться маркер порядка байтов. Это объясняется в следующем разделе.

BOM: полезный крокозябр

Возможно, вы заметили, что в примере 4.4 в начале последовательности в кодировке UTF-16 находились два дополнительных байта. Приведем их еще раз:

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

Речь идет о байтах `b'\xff\xfe'`. Это *BOM* – byte-order mark (маркер порядка байтов). В данном случае он говорит, что порядок байтов прямой, т. е. принятый в процессоре Intel, на котором производилось кодирование.

На машине с прямым порядком байтов для каждой кодовой позиции первым идет младший байт: буква `'E'` с кодовой позицией U+0045 (десятичное 69) представлена в позициях со смещением 2 и 3 от начала последовательности числами `69` и `0`:

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

На машине с обратным порядком байтов кодировка была бы противоположной; буква `'E'` была бы закодирована числами `0` и `69`.

Во избежание недоразумений в начало текстовых файлов в кодировке UTF-16 добавляется специальный невидимый символ НЕРАЗРЫВНЫЙ ПРОБЕЛ НУЛЕВОЙ ШИРИНЫ (U+FEFF). В системе с прямым порядком байтов он кодируется байтами `b'\xff\xfe'` (десятичные 255, 254). Поскольку символ в кодовой позиции U+FFFE не существует – это задумано специально, – последовательность байтов `b'\xff\xfe'` должна означать НЕРАЗРЫВНЫЙ ПРОБЕЛ НУЛЕВОЙ ШИРИНЫ в кодировке с прямым порядком, поэтому кодек знает, каким должен быть порядок байтов.

Существует вариант кодировки UTF-16 – UTF-16LE – специально предназначенный для систем с прямым порядком байтов, а также его аналог для систем с обратным порядком – UTF-16BE. В случае их использования маркер BOM не добавляется:

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

Предполагается, что BOM, если он присутствует, будет отфильтрован кодеком UTF-16, так что останется только сам текст файла без НЕРАЗРЫВНОГО ПРОБЕЛА НУЛЕВОЙ ШИРИНЫ. Стандарт гласит, что для файла в кодировке UTF-16 без маркера BOM следует предполагать кодировку UTF-16BE (с обратным порядком байтов). Однако в архитектуре Intel x86 порядок байтов прямой, поэтому на практике в изобилии встречаются файлы в кодировке UTF-16 с прямым порядком байтов без BOM.

Проблема порядка байтов возникает только для кодировок, в которых символы кодируются словами, состоящими из нескольких байтов, например UTF-16 и UTF-32. Существенное достоинство UTF-8 заключается в том, что эта кодировка порождает одни и те же последовательности байтов вне зависимости от машинной архитектуры, поэтому BOM не нужен. Тем не менее некоторые приложения Windows (и прежде всего Блокнот) добавляют BOM и в файлы в кодировке UTF-8, а для Excel наличие BOM означает, что файл записан в UTF-8, иначе предполагается, что для его кодирования использовалась кодовая страница Windows. Схема кодировки UTF-8 с BOM в Python называется кодеком UTF-8-SIG. Символ U+FEFF в UTF-8-SIG кодируется последовательностью из трех байт `b'\xef\xbb\xbf'`. Поэтому файл, начинающийся такими байтами, скорее всего, закодирован в UTF-8 и содержит BOM.

Замечание Калеба по поводу UTF-8-SIG

Калеб Хэттинг – один из технических рецензентов – предлагает всегда использовать кодек UTF-8-SIG при чтении файлов в кодировке UTF-8. Это безвредно, потому что UTF-8-SIG правильно читает файлы с BOM и без BOM и не возвращает сам символ BOM. При записи файлов я рекомендую использовать UTF-8 ради интероперабельности. Например, Python-скрипт можно сделать исполняемым в системах Unix, добавив в начало комментарий `#!/usr/bin/env python3`. Первые два байта должны быть равны `b'#!'`, иначе этот прием работать не будет, но наличие BOM нарушает соглашение. Если одним из требований является возможность экспорта данных для приложений, ожидающих BOM, используйте UTF-8-SIG, но помните, что написано в документации по кодекам Python (*https://docs.python.org/3/library/codecs.html#encodings-and-unicode*): «В UTF-8 использование BOM не рекомендуется, и в общем случае его следует избегать».

Перейдем теперь к обработке текстовых файлов в Python 3.

Обработка текстовых файлов

На практике обрабатывать текстовые файлы лучше всего, применяя «сэндвич Unicode» (рис. 4.2)[1]. Это означает, что тип `bytes` следует декодировать в `str` на как

[1] Впервые словосочетание «сэндвич Unicode» встретилось мне в замечательном выступлении Нэда Бэтчелдера «Pragmatic Unicode» на конференции US PyCon 2012 (*http://nedbatchelder.com/text/unipain/unipain.html*).

можно более ранних стадиях ввода (например, при открытии файла для чтения). «Котлета» в сэндвиче – это бизнес-логика вашей программы, внутри которой обрабатываются только объекты `str`. Никогда не следует производить кодирование или декодирование в середине обработки. На этапе вывода объекты `str` кодируются в `bytes` как можно позже. Именно так работает большинство веб-каркасов, так что их пользователям редко приходится иметь дело с типом `bytes`. Например, в Django представления должны выводить строки `str`, а Django сам позаботится о кодировании ответа в `bytes`, применяя по умолчанию кодировку UTF-8.

Python 3 облегчает следование этой рекомендации, потому что встроенная функция `open` производит необходимое декодирование при чтении и кодирование при записи файлов в текстовом режиме, т. е. от метода `my_file.read()` мы получаем объекты `str` и их же передаем методу `my_file.write(text)`.

Таким образом, работать с текстовыми файлами просто. Но, всегда полагаясь на кодировку по умолчанию, вы можете горько пожалеть.

Рис. 4.2. Сэндвич Unicode – рекомендуемый способ обработки текста

Взгляните на сеанс оболочки в примере 4.8. Сможете найти ошибку?

Пример 4.8. Проблема платформенно-зависимой кодировки (выполнив этот код на своей машине, вы, возможно, наткнетесь на проблему, а возможно, и нет)

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

Ошибка заключается в том, что я задал кодировку UTF-8 при записи в файл, но забыл сделать это при чтении, поэтому Python предположил, что используется системная кодировка по умолчанию – Windows 1252, – и декодировал два последних байта в файле как символы `'Ã©'` вместо `'é'`.

Я выполнял пример 4.8 на машине под управлением Windows 10 (сборка 18363) с 64-разрядным Python 3.8.1. Те же самые предложения в последних версиях GNU/Linux и Mac OS X работают без ошибок, потому что в них по умолчанию предполагается кодировка UTF-8, и это создает ложное впечатление, будто все хорошо. Если бы мы опустили аргумент `encoding` при открытии файла для записи, то была бы использована местная кодировка по умолчанию, и мы правильно прочитали бы файл в той же самой кодировке. Но тогда этот скрипт генерировал бы файлы с разным байтовым содержимым на разных платфор-

мах и даже при различных настройках локали на одной и той же платформе, и мы получили бы проблему совместимости.

Код, который должен работать на разных машинах или в разных ситуациях, не должен зависеть от кодировки по умолчанию. Всегда явно задавайте аргумент `encoding=` при открытии текстовых файлов, потому что умолчания могут зависеть от машины и даже меняться на одной и той же машине.

В примере 4.8 есть любопытная деталь: функция `write` в первом предложении говорит, что было записано четыре символа, а в следующей строке читается пять символов. В примере 4.9, где приведен расширенный вариант примера 4.8, объясняется этот и другие курьезы.

Пример 4.10. Более пристальное изучение запуска примера 4.9 в Windows вскрывает ошибку и показывает, как ее исправить

```
>>> fp ❶
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
>>> fp.write('café') ❷
4
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size ❸
5
>>> fp2 = open('cafe.txt')
>>> fp2 ❹
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding ❺
'cp1252'
>>> fp2.read() ❻
'cafÃ©'
>>> fp3 = open('cafe.txt', encoding='utf_8') ❼
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read() ❽
'café'
>>> fp4 = open('cafe.txt', 'rb') ❾
>>> fp4 ❿
<_io.BufferedReader name='cafe.txt'>
>>> fp4.read() ⓫
b'caf\xc3\xa9'
```

❶ По умолчанию `open` открывает файл в текстовом режиме и возвращает объект `TextIOWrapper`.

❷ Метод `write` объекта `TextIOWrapper` возвращает количество записанных символов Unicode.

❸ Функция `os.stat` сообщает, что файл содержит 5 байт; в кодировке UTF-8 буква `'é'` представлена двумя байтами: 0xc3 и 0xa9.

❹ В результате открытия текстового файла без явного указания кодировки возвращается объект `TextIOWrapper`, в котором установлена кодировка, взятая из локали.

❺ В объекте `TextIOWrapper` имеется атрибут `encoding`, который можно опросить, в данном случае он равен `cp1252`.

❻ В кодировке Windows `cp1252` байт 0xc3 соответствует символу «Ã» (A с тильдой), а 0xa9 – знаку копирайта.
❼ Открытие того же файла с указанием правильной кодировки.
❽ Ожидаемый результат: те же самые четыре символа Unicode `'café'`.
❾ При задании флага `'rb'` файл открывается в двоичном режиме.
❿ Возвращенный объект имеет тип `BufferedReader`, а не `TextIOWrapper`.
⓫ Чтение возвращает те байты, которые ожидаются.

Не открывайте текстовые файлы в двоичном режиме, если не собираетесь анализировать содержимое файла на предмет определения кодировки, да и в этом случае лучше пользоваться библиотекой Chardet, а не изобретать велосипед (см. раздел «Как определить кодировку последовательности байтов» выше). В обычной программе двоичный режим следует использовать только для открытия двоичных файлов, например растровых изображений.

Проблема, встретившаяся нам в примере 4.9, возникла из-за неверного предположения о кодировке по умолчанию при открытии текстового файла. Как показано в следующем разделе, существует несколько источников таких умолчаний.

Остерегайтесь кодировок по умолчанию

На установку кодировки по умолчанию в Python влияет несколько параметров. См. скрипт *default_encodings.py* в примере 4.10.

Пример 4.10. Исследование кодировок по умолчанию

```
import locale
import sys

expressions = """
        locale.getpreferredencoding()
        type(my_file)
        my_file.encoding
        sys.stdout.isatty()
        sys.stdout.encoding
        sys.stdin.isatty()
        sys.stdin.encoding
        sys.stderr.isatty()
        sys.stderr.encoding
        sys.getdefaultencoding()
        sys.getfilesystemencoding()
    """

my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(f'{expression:>30} -> {value!r}')
```

Этот скрипт выводит одно и то же в GNU/Linux (Ubuntu 14.04 – 19.10) и macOS (10.9–10.14), показывая, что в данных системах всюду используется кодировка `UTF-8`:

```
$ python3 default_encodings.py
 locale.getpreferredencoding() -> 'UTF-8'
                 type(my_file) -> <class '_io.TextIOWrapper'>
              my_file.encoding -> 'UTF-8'
           sys.stdout.isatty() -> True
           sys.stdout.encoding -> 'utf-8'
            sys.stdin.isatty() -> True
            sys.stdin.encoding -> 'utf-8'
           sys.stderr.isatty() -> True
           sys.stderr.encoding -> 'utf-8'
      sys.getdefaultencoding() -> 'utf-8'
   sys.getfilesystemencoding() -> 'utf-8'
```

Однако в Windows выводится нечто совершенно иное (см. пример 4.11).

Пример 4.11. Кодировки по умолчанию в оболочке Windows 10 PowerShell (cmd.exe дает такой же результат)

```
> chcp ❶
Active code page: 437
> python default_encodings.py ❷
locale.getpreferredencoding() -> 'cp1252' ❸
type(my_file) -> <class '_io.TextIOWrapper'>
my_file.encoding -> 'cp1252'    ❹
sys.stdout.isatty() -> True     ❺
sys.stdout.encoding -> 'utf-8' ❻
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'utf-8'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'utf-8'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'utf-8'
```

❶ `chcp` показываетактивную кодовую страницу для консоли: 437.
❷ При запуске *default_encodings.py* с выводом на консоль.
❸ `locale.getpreferredencoding()` – самый важный параметр.
❹ Длятекстовыхфайловпоумолчаниюиспользуется`locale.getpreferredencoding()`.
❺ Вывод производится на консоль, поэтому `sys.stdout.isatty()` равно `True`.
❻ Но `sys.stdout.encoding` не такая же, как кодовая страница для консоли, печатаемая `chcp`!

Поддержка Unicode в самой Windows и в Python для Windows улучшилась за время, прошедшее с выхода первого издания. В Windows 7 с версией Python 3.4 скрипт 4.11 сообщал о четырех разных кодировках. Кодировки для `stdout`, `stdin` и `stderr` совпадали с активной кодовой страницей, напечатанной командой `chcp`, теперь же во всех этих случаях используется `utf-8` – благодаря предложению из документа PEP 528 «Change Windows console encoding to UTF-8» (*https://peps.python.org/pep-0528/*), реализованному в Python 3.6, и поддержке Unicode в PowerShell и в *cmd.exe* (начиная со сборки Windows 1809, вышедшей в октябре 2018 года)[1]. Странно, что `chcp` и `sys.stdout.encoding` печатают разные кодировки, когда `stdout` связан с консолью, но хорошо уже то, что мы наконец-то можем пе-

[1] Источник: «Windows Command-Line: Unicode and UTF-8 Output Text Buffer» (*https://devblogs.microsoft.com/commandline/windows-command-line-unicode-and-utf-8-output-text-buffer/*).

чатать Unicode-строки, не вызывая ошибок кодировки в Windows, – если только пользователь не перенаправит выход в файл, как мы вскоре увидим. Это не значит, что все ваши любимые эмодзи будут видны на консоли; все зависит от шрифта, которым пользуется консоль.

Еще одно изменение, описанное в документе PEP 529 «Change Windows filesystem encoding to UTF-8» (*https://peps.python.org/pep-0529/*) и также реализованное в Python 3.6, изменило кодировку файловой системы (используемую для представления имен каталогов и файлов) с Microsoft MBCS на UTF-8.

Однако если перенаправить вывод примера 4.10 в файл:

```
Z:\>python default_encodings.py > encodings.log
```

то `sys.stdout.isatty()` становится равным `False` и `sys.stdout.encoding` устанавливается путем обращения к `locale.getpreferredencoding()`, т. е. `'cp1252'` на данной машине. Но `sys.stdin.encoding` и `sys.stderr.encoding` остаются равными `utf-8`.

В примере 4.12 я использую управляющую последовательность `'\N{}'` для литералов Unicode, при этом внутри скобок записывается официальное название символа. Это довольно многословно, но зато явно и безопасно: Python возбуждает исключение `SyntaxError`, если указанное имя не существует – гораздо лучше, чем записывать шестнадцатеричное число, которое может оказаться неверным, но узнаете вы об этом гораздо позже. Да и, скорее всего, вы все равно написали бы комментарий, объясняющий назначение кодов, так что с пространностью `\N{}` легко смириться.

Это означает, что скрипт, подобный приведенному в примере 4.12, работает при выводе на консоль, но может ломаться при перенаправлении вывода в файл.

Пример 4.12. stdout_check.py

```
import sys
from unicodedata import name

print(sys.version)
print()
print('sys.stdout.isatty():', sys.stdout.isatty())
print('sys.stdout.encoding:', sys.stdout.encoding)
print()

test_chars = [
    '\N{HORIZONTAL ELLIPSIS}',      # есть в cp1252, нет в cp437
    '\N{INFINITY}',                 # есть в cp437, нет в cp1252
    '\N{CIRCLED NUMBER FORTY TWO}', # нет ни в cp437, ни в cp1252
]
for char in test_chars:
    print(f'Trying to output {name(char)}:')
    print(char)
```

Пример 4.12 выводит результат `sys.stdout.isatty()`, значение `sys.stdout.encoding` и следующие три символа:

- `'…' HORIZONTAL ELLIPSIS` – есть в кодировке CP 1252, но не в CP 437;
- `'∞' INFINITY` – есть в кодировке CP 437, но не в CP 1252;
- `'㊷' CIRCLED NUMBER FORTY TWO` – нет ни в cp437, ни в cp1252.

При запуске `stdout_check.py` в PowerShell или *cmd.exe* он работал, как показано на рис. 4.3.

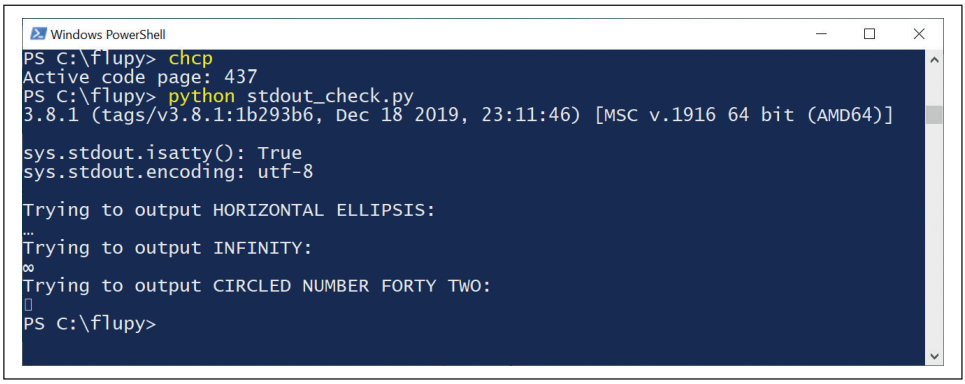


Рис. 4.3. Выполнение stdout_check.py в PowerShell

Хотя `chcp` сообщает, что активна кодовая страница 437, кодировка `sys.stdout.encoding` равна UTF-8, поэтому символы `HORIZONTAL ELLIPSIS` и `INFINITY` выводятся правильно. Символ `CIRCLED NUMBER FORTY TWO` заменен прямоугольником, но ошибки не возникло. Вероятно, символ-то допустимый, но в консольном шрифте нет соответствующего ему глифа.

Однако после перенаправления вывода *stdout_check.py* в файл я получил картину, показанную на рис. 4.4.

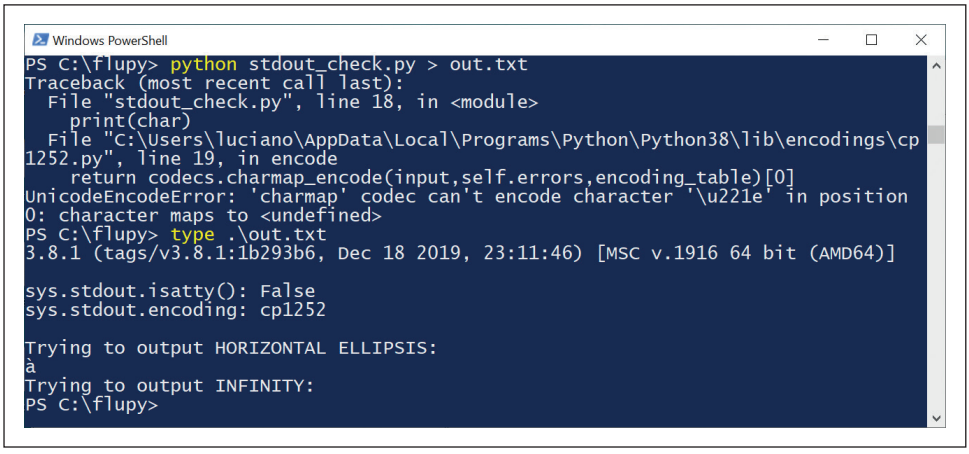


Рис. 4.4. Выполнение stdout_check.py в PowerShell с перенаправлением вывода

Первая из показанных на рис. 4.4 проблем – ошибка `UnicodeEncodeError`, в которой упомянут символ `'\u221e'`. Она возникла, потому что кодировка `sys.stdout.encoding` равна `'cp1252'`, а в этой кодовой странице нет символа `INFINITY`.

Прочитав файл *out.txt* командой `type` или открыв его в редакторе, например VS Code или Sublime Text, мы увидим, что вместо символа `HORIZONTAL ELLIPSIS` присутствует символ `'à'` (`LATIN SMALL LETTER A WITH GRAVE`). Оказывается, что значе-

ние байта 0x85 в кодировке CP 1252 означает '…', а в кодировке CP 437 – символ 'à'. Так что, похоже, активная кодовая страница играет роль – ни разумной, ни полезной ее не назовешь, но частично она объясняет проблемы с Unicode.

Для описанных экспериментов я пользовался ноутбуком, сконфигурированным для рынка США с установленной ОС Windows 10 OEM. В версиях Windows, локализованных для других стран, кодировка может быть настроена иначе. Например, в Бразилии на консоли Windows по умолчанию используется кодовая страница 850, а не 437.

Подводя итоги этой приводящей в исступление ситуации с кодировками по умолчанию, в последний раз рассмотрим различные кодировки, встречающиеся в примере 4.11.

- Если опустить аргумент `encoding` при открытии файла, то умолчание определяется методом `locale.getpreferredencoding()` (`'cp1252'` в примере 4.11).
- До версии Python 3.6 кодировка `sys.stdout/stdin/stderr` определялась переменной окружения `PYTHONIOENCODING` (*https://docs.python.org/3/using/cmdline.html#envvar-PYTHONIOENCODING*). Теперь она игнорируется, если только значением переменной `PYTHONLEGACYWINDOWSSTDIO` (*https://docs.python.org/3/using/cmdline.html#envvar-PYTHONLEGACYWINDOWSSTDIO*) не является непустая строка, а если является, то кодировка стандартного ввода-вывода равна UTF-8, когда поток связан с консолью, но совпадает с `locale.getpreferredencoding()`, когда поток перенаправлен на файл.
- Функция `sys.getdefaultencoding()` используется самим интерпретатором Python для неявных преобразований двоичных данных в строку и обратно. Изменение этого параметра не поддерживается.
- Функция `sys.getfilesystemencoding()` применяется для кодирования и декодирования имен файлов (но не их содержимого). Она вызывается, когда `open()` получает имя файла в виде строки `str`; если же имя файла задано аргументом типа `bytes`, то оно передается API операционной системы без изменения.

В GNU/Linux и macOS все эти кодировки по умолчанию совпадают с UTF-8, и такое положение существует уже несколько лет, поэтому подсистема ввода-вывода обрабатывает все символы Unicode. В Windows не только используются различные кодировки в одной и той же системе, но обычно это еще и кодовые страницы, например `'cp850'` или `'cp1252'`, которые поддерживают только ASCII и еще 127 символов, отличающихся в разных кодировках. Поэтому у пользователей Windows гораздо больше шансов столкнуться с ошибками кодирования при малейшей небрежности.

Подводя итоги, можно сказать, что самым важным из всех относящихся к кодировкам параметров является значение, возвращаемое методом `locale.getpreferredencoding()`: оно подразумевается по умолчанию при открытии текстовых файлов и при вводе-выводе на `sys.stdout/stdin/stderr`, если поток перенаправлен в файл. Однако в документации (*https://docs.python.org/3/library/ locale.html#locale.getpreferredencoding*) мы читаем:

`locale.getpreferredencoding(do_setlocale=True)`

> Вернуть кодировку, используемую для текстовых данных, в соответствии с предпочтениями пользователя. Предпочтения задаются по-разному в разных системах и не всегда доступны из программы, поэтому данная функция возвращает только предположительное значение [...].

Таким образом, лучшее, что можно посоветовать в части кодировок по умолчанию: не полагайтесь на них.

Если вы будете поступать, как рекомендует сэндвич Unicode, и всегда явно указывать кодировку, то избежите множества неприятностей. К сожалению, проблемы работы с Unicode не заканчиваются, даже если вы правильно преобразуете `bytes` в `str`. В следующих двух разделах рассматриваются темы, которые не вызывают ни малейших трудностей в краю ASCII, но становятся весьма сложными на планете Unicode: нормализация текста (т. е. приведение его к единому представлению для сравнения) и сортировка.

Нормализация Unicode для надежного сравнения

Сравнение строк осложняется тем, что в Unicode есть модифицирующие символы: диакритические и другие знаки, присоединяемые к предыдущему символу, так что при печати оба символа выглядят как единое целое.

Например, слово «café» можно составить двумя способами, из четырех или из пяти кодовых позиций, хотя результат будет выглядеть одинаково:

```
>>> s1 = 'café'
>>> s2 = 'cafe\N{COMBINING ACUTE ACCENT}'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

Кодовая позиция U+0301 называется COMBINING ACUTE ACCENT (МОДИФИЦИРУЮЩИЙ АКУТ). Если она следует за «e», то результат отображается как «é». В стандарте Unicode последовательности вида `'é'` и `'e\u0301'` называются «каноническими эквивалентами» и предполагается, что приложения будут считать их одинаковыми. Но Python видит две разные последовательности кодовых позиций и одинаковыми их не считает.

Решение состоит в том, чтобы использовать функцию `unicodedata.normalize`. Первым аргументом функции передается одна из четырех строк: `'NFC'`, `'NFD'`, `'NFKC'` или `'NFKD'`. Сначала рассмотрим первые две.

Форма нормализации C (NFC) производит композицию двух кодовых позиций с целью получения самой короткой эквивалентной строки, а форма нормализации D (NFD) производит декомпозицию, т. е. разложение составного символа на базовый и модифицирующие. В результате выполнения обеих нормализаций сравнение работает, как и ожидается:

```
>>> from unicodedata import normalize
```

```
>>> s1 = 'café'
>>> s2 = 'cafe\N{COMBINING ACUTE ACCENT}'
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
```

Драйверы клавиатуры обычно генерируют составные символы, поэтому набранный пользователем текст по умолчанию оказывается в формате NFC. Но для пущей уверенности лучше прогнать строки через `normalize('NFC', user_text)` перед сохранением. Форма нормализации NFC рекомендуется также консорциумом W3C в документе «Character Model for the World Wide Web: String Matching and Searching (*http://www.w3.org/TR/charmod-norm/*).

Некоторые одиночные символы форма NFC преобразует в другие одиночные символы. Символ ома, единицы электрического сопротивления, Ω преобразуется в греческую букву омега в верхнем регистре. Визуально они ничем не отличаются, но при сравнении не совпадают, поэтому во избежание сюрпризов необходимо производить нормализацию:

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

В акронимах двух других форм нормализации – NFKC и NFKD – буква K означает «compatibility» (совместимость). Это более строгие формы нормализации, затрагивающие так называемые «символы совместимости». Хотя одна из целей Unicode – определить единственную «каноническую» кодовую позицию для каждого символа, некоторые символы встречаются несколько раз ради совместимости с предшествующими стандартами. Например, знак «микро», `MICRO SIGN`, `'µ' (U+00B5)` был добавлен в Unicode для поддержки обратимого преобразования в `latin1`, хотя тот же самый символ является также частью греческого алфавита, где ему соответствует кодовая позиция `U+03BC` (`GREEK SMALL LETTER MU`, `СТРОЧНАЯ ГРЕЧЕСКАЯ БУКВА МЮ`). Поэтому знак «микро» считается «символом совместимости».

В формах NFKC и NFKD каждый символ совместимости заменяется «совместимой декомпозицией» из одного или более символов, которая считается «предпочтительным» представлением, даже если при этом возникает потеря форматирования – в идеале форматирование должно быть функцией внешней разметки, а не частью Unicode. Например, совместимой декомпозицией дроби «одна вторая» `'½' (U+00BD)` является последовательность трех символов `'1/2'`,

а совместимой декомпозицией знака «микро» 'µ' (U+00B5) – строчная буква мю 'μ' (U+03BC)[1].

Вот как NFKC работает на практике:

```
>>> from unicodedata import normalize, name
>>> half = '\N{VULGAR FRACTION ONE HALF}'
>>> print(half)
½ >>> normalize('NFKC', half)
'1/2'
>>> for char in normalize('NFKC', half):
...     print(char, name(char), sep='\t')
...
1       DIGIT ONE
/       FRACTION SLASH
2       DIGIT TWO
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'µ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('µ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')
```

В то время как '1/2' – разумная замена для '½', а знак «микро» действительно совпадает со строчной греческой буквой мю, преобразование '4²' в '42' изменяет смысл. Приложение могло бы сохранить '4²' как '4²', но функция normalize ничего не знает о форматировании. Поэтому формы NFKC и NFKD могут терять или искажать информацию, но в то же время дают удобное промежуточное представление для поиска и индексирования.

К сожалению, в Unicode все всегда сложнее, чем кажется. Для символа VULGAR FRACTION ONE HALF (ПРОСТАЯ Дробь Одна Вторая) форма нормализации NFKC порождает символы 1 и 2, соединенные символом FRACTION SLASH (ДРОБНАЯ ЧЕРТА), а не SOLIDUS (КОСАЯ ЧЕРТА) – знакомым символом с десятичным кодом ASCII 47. Поэтому поиск последовательности трех символов ASCII '1/2' не найдет нормализованной последовательности Unicode.

Формы нормализации NFKC и NFKD следует применять с осторожностью и только в особых случаях, например для поиска и индексирования, а не для постоянного хранения текста, поскольку выполняемые ими преобразования могут приводить к потере данных.

Для подготовки текста к поиску или индексированию полезна еще одна операция: сворачивание регистра. Это и есть тема следующего раздела.

[1] Любопытно, что знак «микро» считается символом совместимости, а знак «ом» нет. В результате NFC не трогает знак «микро», но изменяет знак «ом» на заглавную букву омега, тогда как NFKC и NFKD заменяют и «ом», и «микро» символами греческого алфавита.

Сворачивание регистра

Сворачивание регистра – это, по существу, перевод всего текста в нижний регистр с некоторыми дополнительными преобразованиями. Для этой цели предназначен метод `str.casefold()`.

Если строка `s` содержит только символы из набора `latin1`, то `s.casefold()` дает такой же результат, как `s.lower()`, с двумя исключениями: знак «микро» `'µ'` заменяется строчной греческой буквой мю (в большинстве шрифтов они выглядят одинаково), а немецкая «эсцет» преобразуется в «ss»:

```
>>> micro = 'µ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('µ', 'µ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
```

Существует 300 кодовых позиций, для которых `str.casefold()` и `str.lower()` дают различные результаты.

Как все связанное с Unicode, сворачивание регистра – сложная лингвистическая проблема со множеством особых случаев, но разработчики ядра Python приложили максимум усилий, чтобы предложить решение, устраивающее большинство пользователей.

В следующих двух разделах мы применим знания о нормализации к разработке служебных функций.

Служебные функции для сравнения нормализованного текста

Как мы видели, формы нормализации NFC и NFD безопасны и позволяют достаточно осмысленно сравнивать Unicode-строки. Для большинства приложений NFC – наилучшая нормализованная форма. Для сравнения строк без учета регистра предназначен метод `str.casefold()`.

Если вы работаете с текстами на многих языках, рекомендуем включить в свой арсенал функции наподобие `nfc_equal` и `fold_equal`, показанные в примере 4.13.

Пример 4.13. normeq.py: сравнение нормализованных Unicode-строк

```
"""
Служебные функции для сравнения нормализованных Unicode-строк.

Использование нормальной формы C, с учетом регистра:

    >>> s1 = 'cafe'
    >>> s2 = 'cafe\u0301'
    >>> s1 == s2
```

```
    False
    >>> nfc_equal(s1, s2)
    True
    >>> nfc_equal('A', 'a')
    False

Использование нормальной формы C, со сворачиванием регистра:

    >>> s3 = 'Strasse'
    >>> s4 = 'strasse'
    >>> s3 == s4
    False
    >>> nfc_equal(s3, s4)
    False
    >>> fold_equal(s3, s4)
    True
    >>> fold_equal(s1, s2)
    True
    >>> fold_equal('A', 'a')
    True

"""

from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

Помимо нормализации и сворачивания регистра (то и другое – части стандарта Unicode), иногда бывают полезны более глубокие преобразования, например `'café'` в `'cafe'`. В следующем разделе мы покажем, когда это необходимо и как делается.

Экстремальная «нормализация»: удаление диакритических знаков

Секретный рецепт поиска Google скрывает много разных хитростей, одна из них – полное игнорирование диакритических знаков (акцентов, седилей и т. д.), по крайней мере в некоторых контекстах. Удаление диакритических знаков, строго говоря, не является нормализацией, потому что зачастую при этом меняется смысл слов и поиск может находить не то, что нужно. Но жизнь есть жизнь: многие ленятся ставить диакритические знаки или не знают, как это нужно делать, да и правила правописания время от времени меняются. Игнорирование диакритики помогает справиться с этими проблемами.

Но даже если оставить поиск в стороне, удаление диакритических знаков делает URL-адреса более удобочитаемыми, по крайней мере в языках на основе латиницы. Взгляните на URL статьи Википедии о городе Сан-Паулу:

```
http://en.wikipedia.org/wiki/S%C3%A3o_Paulo
```

Часть `%C3%A3` – результат URL-кодирования представления буквы «ã» (а с тильдой) в кодировке UTF-8. Показанный ниже URL гораздо понятнее, пусть даже правописание в нем хромает:

```
http://en.wikipedia.org/wiki/Sao_Paulo
```

Для удаления всех диакритических знаков из `str` можно воспользоваться функцией из примера 4.14.

Пример 4.14. Функция для удаления всех модифицирующих символов (модуль sanitize.py)

```
import unicodedata
import string

def shave_marks(txt):
    """Remove all diacritic marks"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    shaved = ''.join(c for c in norm_txt
                     if not unicodedata.combining(c)) ❷
    return unicodedata.normalize('NFC'shaved) ❸
```

❶ Разложить все символы на базовые и модифицирующие.
❷ Найти все модифицирующие символы.
❸ Произвести обратную композицию.

В примере 4.15 демонстрируются два применения функции `shave_marks`.

Пример 4.15. Два применения функции `shave_marks` из примера 4.14

```
>>> order = '"Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of açaí."'
>>> shave_marks(order)
'"Herr Voß: • ½ cup of OEtker™ caffe latte • bowl of acai."' ❶
>>> Greek = 'Zéφυρος, Zéfiro'
>>> shave_marks(Greek)
'Zεφυρος, Zefiro ❷
```

❶ Заменены только буквы «è», «ç» и «í».
❷ Заменены буквы «é» и «é».

Функция `shave_marks` работает правильно, но, быть может, чрезмерно усердствует. Часто диакритические знаки удаляются только для того, чтобы перевести текст из кодировки Latin в чистый ASCII, но `shave_marks` изменяет также и нелатинские символы, например греческие буквы, которые – что с акцентами, что без – никогда не превратятся в ASCII. Поэтому имеет смысл проанализировать каждый базовый символ и удалять присоединенные знаки, только если он является буквой из набора символов Latin. Именно это делает функция из примера 4.16.

Пример 4.16. Функция удаления модифицирующих знаков только для символов из набора Latin (предложения импорта опущены, поскольку это часть модуля sanitize.py из примера 4.14)

```
def shave_marks_latin(txt):
    """Удалить все диакритические знаки для базовых символов набора Latin"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    latin_base = False
    preserve = []
```

```
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base: ❷
            continue # игнорировать диакритические знаки
                     # для базовых символов набора Latin
        preserve.append(c) ❸
        # если это не модифицирующий символ, значит, новый базовый
        if not unicodedata.combining(c): ❹
            latin_base = c in string.ascii_letters
    shaved = ''.join(preserve)
    return unicodedata.normalize('NFC'shaved) ❺
```

❶ Разложить все символы на базовые и модифицирующие.
❷ Пропустить модифицирующие символы, если базовый из набора Latin.
❸ В противном случае сохранить текущий символ.
❹ Распознать новый базовый символ и определить, принадлежит ли он набору Latin.
❺ Произвести обратную композицию.

Еще более радикальный шаг – заменить часто встречающиеся в западных текстах символы (например, фигурные кавычки, длинные тире, маркеры списков и т. д.) эквивалентными символами из набора ASCII. Этим занимается функция `asciize` из примера 4.17.

Пример 4.17. Преобразование некоторых западных типографических символов в ASCII (этот код также входит составной частью в модуль sanitize.py из примера 4.14)

```
single_map = str.maketrans("""‚ƒ„ˆ‹''""•--˜›""", ❶
                           """'ƒ"^<''""---~>""")
multi_map = str.maketrans({ ❷
    '€': 'EUR',
    '…': '...',
    'Æ': 'AE',
    'æ': 'ae',
    'OE': 'OE',
    'oe': 'oe',
    '™': '(TM)',
    '%': '<per mille>',
    '†': '**',
    '‡': '***',
})
multi_map.update(single_map) ❸

def dewinize(txt):
    """Заменить символы Win1252 символами или последовательностями символов ASCII """
    return txt.translate(multi_map) ❹

def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt)) ❺
    no_marks = no_marks.replace('ß', 'ss') ❻
    return unicodedata.normalize(NFK C',nomarks) ❼
```

❶ Построить таблицу соответствия для замены одного символа другим.
❷ Построить таблицу соответствия для замены символа строкой символов.
❸ Объединить таблицы соответствия.

❹ Функция `dewinize` не изменяет символы из наборов `ASCII` и `latin1`, а затрагивает только добавления к `latin1`, включенные Microsoft в набор `cp1252`.
❺ Применить `dewinize` и удалить диакритические знаки.
❻ Заменить эсцет на «ss» (мы не пользуемся сворачиванием регистра, потому что хотим сохранить исходный регистр).
❼ Применить нормализацию NFKC для композиции символов с их кодовыми позициями совместимости.

В примере 4.18 показана функция `asciize` в действии.

Пример 4.18. Два применения функции `asciize` из примера 4.17

```
>>> order = '"Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of açaí."'
>>> dewinize(order)
'"Herr Voß: - ½ cup of OEtker(TM) caffè latte - bowl of açaí."' ❶
>>> asciize(order)
'"Herr Voss: - 1/2 cup of OEtker(TM) caffe latte - bowl of acai."' ❷
```

❶ `dewinize` заменяет фигурные кавычки, маркеры списка и знак торговой марки ™.
❷ `asciize` вызывает `dewinize`, затем удаляет диакритические знаки и заменяет `'ß'`.

В разных языках правила удаления диакритических знаков различны. Например, немцы заменяют `'ü'` на `'ue'`. Наша функция `asciize` не настолько рафинирована, поэтому может статься, что к вашему языку она не применима. Впрочем, для португальского работает отлично.

Итак, функции из модуля *sanitize.py* не ограничиваются стандартной нормализацией, а подвергают текст серьезной хирургической операции, которая вполне может изменить его смысл. Лишь обладая знаниями о целевом языке, потенциальных пользователях и способах использования преобразованного текста, можно решить, стоит ли заходить так далеко. А кому все это знать, как не вам?

На этом мы подводим черту под обсуждением нормализации Unicode-текстов. Далее нам предстоит заняться проблемой сортировки.

Сортировка Unicode-текстов

Python сортирует последовательности любого типа, сравнивая элементы один за другим. Для строк это означает сравнение кодовых позиций. Увы, результат получится никуда не годным, если только вы не ограничиваетесь символами ASCII.

Рассмотрим сортировку списка фруктов, произрастающих в Бразилии.

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```

Правила сортировка зависят от локали, но в португальском и многих других языках, основанных на латинице, акценты и седили редко учитываются при

сортировке[1]. Поэтому «cajá» сортируется так же, как «caja» и, следовательно, предшествует «caju».

Отсортированный список `fruits` должен выглядеть так:

```
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

Стандартный способ сортировки не-ASCII текстов в Python – функция `locale.strxfrm`, которая, как написано в документации по модулю `locale` (*https://docs.python.org/3/library/locale.html?highlight=strxfrm#locale.strxfrm*), «преобразует строку, так чтобы ее можно было использовать в сравнениях с учетом локали».

Чтобы можно было воспользоваться функцией `locale.strxfrm`, необходимо сначала установить локаль, отвечающую нуждам приложения, и надеяться, что ОС ее поддерживает. В системах на базе GNU/Linux (Ubuntu 14.04) при выборе локали `pt_BR` нужный результат дает последовательность команд в примере 4.19.

Пример 4.19. locale_sort.py: использование функции `locale.strxfrm` в качестве ключа сортировки

```
import locale
my_locale = locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
print(my_locale)
fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
sorted_fruits = sorted(fruits, key=locale.strxfrm)
print(sorted_fruits)
```

При выполнении примера 4.19 в GNU/Linux (Ubuntu 19.10) с установленной локалью `pt_BR.UTF-8` я получил правильный результат:

```
'pt_BR.UTF-8'
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

Таким образом, до использования `locale.strxfrm` в качестве значения параметра `key` необходимо вызвать `setlocale(LC_COLLATE, «ваша_локаль»)`.

Однако есть несколько подводных камней.

Поскольку установка локали – глобальное действие, вызывать `setlocale` в библиотеке не рекомендуется. Приложение или каркас должны установить локаль в начале работы процесса и затем уже не менять.

- Локаль должна быть установлена в ОС, иначе вызов `setlocale` возбуждает исключение `locale.Error: unsupported locale setting`.
- Необходимо точно знать, как пишется имя локали.
- Локаль должна быть правильно реализована производителем ОС. С Ubuntu 14.04 мне повезло, а с macOS 10.14 – нет. В macOS обращение `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` честно возвращало строку `'pt_BR.UTF-8'`. Однако вызов `sorted(fruits, key=locale.strxfrm)` давал тот же неправильный результат, что и `sorted(fruits)`. Я пробовал также локали `fr_FR`, `es_ES` и `de_DE` в macOS, но ни разу `locale.strxfrm` не отработала, как положено[2].

[1] Диакритические знаки оказывают влияние на сортировку в тех редких случаях, когда слова только ими и отличаются, в подобном случае слово с диакритическим знаком следует за словом без такового.

[2] Я не смог найти решение, но другие тоже жаловались на эту проблему. Алекс Мартелли, один из технических рецензентов, не сталкивался с ошибкой при использовании `setlocale` и `locale.strxfrm` на своем macOS 10.9. Короче говоря, как повезет.

Таким образом, содержащееся в стандартной библиотеке решение для интернационализированной сортировки работает, но лучше всего поддержано в GNU/Linux (или в Windows, если вы специалист по этой ОС). Но даже в этом случае оно зависит от настройки локали, что может вызвать неприятности при развертывании.

По счастью, существует более простое решение: библиотека *pyuca*, доступная на сайте *PyPI*.

Сортировка с помощью алгоритма упорядочивания Unicode

Джеймсу Тауберу (James Tauber), автору многих проектов для Django, должно быть, надоела эта путаница, и он написал модуль PyUCA (*https://pypi.python.org/pypi/pyuca/*), реализацию алгоритма упорядочивания Unicode (Unicode Collation Algorithm – UCA) на чистом Python. В примере 4.20 показано, как просто его использовать.

Пример 4.20. Использование метода `pyuca.Collator.sort_key`

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

Метод удобный и работает правильно в системах GNU/Linux, macOS и Windows, по крайней мере на моей маленькой выборке.

Библиотека *pyuca* не обращает внимания на локаль. Если требуется изменить порядок сортировки, укажите путь к своей таблице упорядочения при вызове конструктора `Collator()`. Оригинальная библиотека пользуется файлом `allkeys.txt` (*https://github.com/jtauber/pyuca*), включенным в дистрибутив. Это просто копия стандартной таблицы элементов упорядочения Unicode (*http://www.unicode.org/Public/UCA/6.3.0/allkeys.txt*) с сайта *Unicode.org*.

PYICU: рекомендация Миро касательно сортировки в UNICODE

(Технический рецензент Мирослав Седивы – полиглот и эксперт по Unicode. Вот что он написал по поводу *pyuca*.)

В *pyuca* имеется один алгоритм сортировки, который не учитывает порядок сортировки в конкретных языках. Например, в немецком буква Ä расположена между A и B, а в шведском она идет после Z. Обратите внимание на расширение PyICU (*https://pypi.org/project/PyICU/*), которое работает как локаль, но без изменения локали процесса. Оно также необходимо, если требуется изменить регистр iİ/ıI в турецком языке. Расширение PyICU необходимо компилировать, поэтому в некоторых системах установить его сложнее, чем библиотеку *pyuca*, написанную на чистом Python.

Кстати говоря, таблица упорядочения – лишь одна из многих составных частей базы данных Unicode, о которой мы поговорим в следующем разделе.

База данных Unicode

В стандарте Unicode приведена целая база данных – в виде многочисленных структурированных текстовых файлов, – которая включает не только сопоставление имен символов кодовым позициям, но также метаданные отдельных символов и информацию о связях между ними. Например, в базе данных Unicode указано, имеет ли символ графическое начертание, является ли он буквой, десятичной цифрой или еще каким-то числовым символом. На основе этой информации работают методы `isidentifier`, `isprintable`, `isdecimal` и `isnumeric` класса `str`. Метод `str.casefold` также пользуется информацией из базы данных Unicode.

Функция `unicodedata.category(char)` возвращает двухбуквенную категорию символа из базы данных Unicode. Методы класса `str` более высокого уровня использовать проще. Например, `label.isalpha()` возвращает `True`, если все символы `label` принадлежат одной из следующих категорий: `Lm`, `Lt`, `Lu`, `Ll` или `Lo`. О том, что означают эти коды, см. раздел «General Category» в статье из англоязычной части Википедии «Unicode character property» (*https://en.wikipedia.org/wiki/Unicode_character_property*).

Поиск символов по имени

В модуле `unicodedata` имеются функции, возвращающие метаданные символов, в т. ч. `unicodedata.name()`, которая возвращает официальное название символа по стандарту. На рис. 4.5 показана эта функция в действии[1].

```
>>> from unicodedata import name
>>> name('A')
'LATIN CAPITAL LETTER A'
>>> name('ã')
'LATIN SMALL LETTER A WITH TILDE'
>>> name('♕')
'BLACK CHESS QUEEN'
>>> name('😸')
'GRINNING CAT FACE WITH SMILING EYES'
```

Рис. 4.5. Исследование `unicodedata.name()` на консоли Python

Функцию `name()` можно использовать в приложениях, позволяющих искать символы по имени. На рис. 4.6 показан результат работы командного скрипта *cf.py*, который принимает в качестве аргументов одно или несколько слов и выводит символы Unicode, в официальных названиях которых встречаются эти слова. Полный исходный код скрипта *cf.py* приведен в примере 4.21.

[1] Это картинка, а не листинг, потому что цифровые инструменты подготовки к печати, применяемые в издательстве O'Reilly, плохо поддерживают эмодзи.

```
$ ./cf.py cat smiling
U+1F638 😸      GRINNING CAT FACE WITH SMILING EYES
U+1F63A 😺      SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 😻      SMILING CAT FACE WITH HEART-SHAPED EYES
```

Рис. 4.6. Использование скрипта cf.py для поиска улыбающихся котят

Поддержка эмодзи сильно зависит от операционной системы и приложения. В последние годы лучшую поддержку эмодзи на терминалах предлагает macOS, за ней следуют современные графические терминалы GNU/Linux. В Windows cmd.exe и PowerShell теперь поддерживают вывод Unicode, но в январе 2020 года, когда я пишу эти строки, они еще не отображали эмодзи, по крайней мере не «из коробки». Технический рецензент Леонардо Рохаэль говорил мне о новом терминале Windows с открытым исходным кодом от Microsoft, который, возможно, предлагает улучшенную поддержку Unicode, чем старые консоли Microsoft. Но у меня не было времени попробовать его.

В примере 4.21 обратите внимание на предложение `if` в функции `find`. В нем используется метод `.issubset()` для быстрой проверки того, встречаются ли все слова, перечисленные в множестве `query`, в списке слов, построенном по имени символа. Благодаря развитому API работы с множествами нам не нужно писать вложенный цикл `for` и еще одно предложение `if` для реализации этой проверки.

Пример 4.21. cf.py: утилита для поиска символов

```
#!/usr/bin/env python3
import sys
import unicodedata

START, END = ord(' '), sys.maxunicode + 1          ❶

def find(*query_words, start=START, end=END):      ❷
    query = {w.upper() for w in query_words}       ❸
    for code in range(start, end):
        char = chr(code)                           ❹
        name = unicodedata.name(char, None)        ❺
        if name and query.issubset(name.split()):  ❻
            print(f'U+{code:04X}\t{char}\t{name}') ❼

def main(words):
    if words:
        find(*words)
    else:
        print('Please provide words to find.')

if __name__ == '__main__':
    main(sys.a rgv[ 1:]
```

❶ Задать умолчания для просматриваемого диапазона кодовых позиций.

❷ `find` принимает аргумент `query_words` и необязательные чисто именованные аргументы, чтобы ограничить диапазон поиска и упростить тестирование.

❸ Преобразовать `query_words` в множество строк в верхнем регистре.
❹ Получить символ Unicode, соответствующий `code`.
❺ Получить имя символа или `None`, если кодовой позиции не назначен никакой символ.
❻ Если имя существует, разбить его на список слов, а затем проверить, что множество `query` является подмножеством этого списка.
❼ Напечатать строку, содержащую кодовую позицию в формате `U+9999`, сам символ и его имя.

В модуле `unicodedata` есть и другие интересные функции. Далее мы рассмотрим несколько функций для получения информации о символах, связанных с числами.

Символы, связанные с числами

В модуле `unicodedata` имеются функции, которые проверяют, представляет ли символ Unicode число, и если да, то возвращают его значение, интересное человеку, а не просто номер кодовой позиции. В примере 4.22 показано использование функций `unicodedata.name()` и `unicodedata.numeric()`, а также методов `.isdecimal()` и `.isnumeric()` класса `str`.

Пример 4.22. Демонстрация работы с метаданными числовых символов в базе данных Unicode (числовые маркеры описывают отдельные столбцы распечатки)

```
import unicodedata
import re

re_digit = re.compile(r'\d')

sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'

for char in sample:
    print(f'U+{ord(char):04x}',                    ❶
        char.center(6),                            ❷
        're_dig' if re_digit.match(char) else '-', ❸
        'isdig' if char.isdigit() else '-',        ❹
        'isnum' if char.isnumeric() else '-',      ❺
        f'{unicodedata.numeric(char):5.2f}',       ❻
        unicodedata.name(char),                    ❼
        sep='\t')
```

❶ Кодовая позиция в формате `U+0000`.
❷ Символ, центрированный в поле длины 6.
❸ Вывести `re_dig`, если символ соответствует регулярному выражению `r'\d'`.
❹ Вывести `isdig`, если `char.isdigit()` равно `True`.
❺ Вывести `isnum`, если `char.isnumeric()` равно `True`.
❻ Числовое значение в поле шириной 5 с двумя десятичными знаками после запятой.
❼ Имя символа Unicode.

В результате выполнения этого скрипта получается распечатка, показанная на рис. 4.7 (при условии что шрифт терминала содержит все эти глифы).

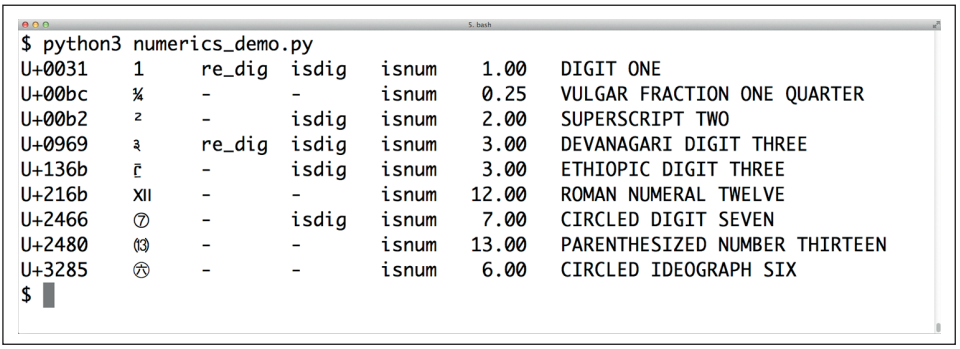


Рис. 4.7. Числовые символы и их метаданные в терминале macOS; `re_dig` означает, что символ соответствует регулярному выражению `r'\d'`

Шестая колонка на рис. 4.7 содержит результат вызова `unicodedata.numeric(char)` для символа. Эта функция говорит о том, что Unicode знает числовые значения символов, представляющих числа. Так что если вы собираетесь написать программу для электронной таблицы, поддерживающей тамильские или римские цифры, вперед и с песней!

По рис. 4.7 видно, что регулярному выражению `r'\d'` соответствует цифра «1» и цифра 3 письменности Деванагари, но не некоторые другие символы, которые функция `isdigit` считает цифрами. Модуль `re` знает о Unicode меньше, чем должен бы. Новый модуль `regex`, доступный на сайте PyPI, имеет целью полностью заменить `re` и поддерживает Unicode лучше[1]. Мы вернемся к модулю `re` в следующем разделе.

В этой главе мы пользовались несколькими функциями из модуля `unicodedata`, но на самом деле их гораздо больше. См. описание модуля `unicodedata` в документации по стандартной библиотеке (*https://docs.python.org/3/library/unicodedata.html*).

Далее мы кратко познакомимся с двухрежимным API, предоставляющим функции, которые принимают в качестве аргументов `str` и `bytes` и работают по-разному в зависимости от типа.

Двухрежимный API

В стандартной библиотеке есть функции, которые принимают в качестве аргументов значения типа `str` или `bytes` и ведут себя по-разному в зависимости от типа. Примеры имеются в модулях `re` и `os`.

str и bytes в регулярных выражениях

Если при построении регулярного выражения был задан аргумент типа `bytes`, то образцам вида `\d` или `\w` будут соответствовать только ASCII-символы. Наоборот, если был задан аргумент типа `str`, то этим образцам будут соответствовать цифры и буквы в смысле Unicode, а не только ASCII. В примере 4.23 и на рис. 4.8 показано сопоставление букв, ASCII-цифр, надстрочных индексов и тамильских цифр с образцами типа `str` и `bytes`.

[1] Хотя цифры он распознавал ничуть не лучше модуля `re`.

Пример 4.23. ramanujan.py: сравнение поведения простых регулярных выражений с аргументами типа `str` и `bytes`

```
import re

re_numbers_str = re.compile(r'\d+') ❶
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+') ❷
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef" ❸
            " as 1729 = 1³ + 12³ = 9³ + 10³.") ❹

text_bytes = text_str.encode('utf_8') ❺

print(f'Text\n {text_str!r}')
print('Numbers')
print(' str :', re_numbers_str.findall(text_str))      ❻
print(' bytes:', re_numbers_bytes.findall(text_bytes)) ❼
print('Words')
print(' str :', re_words_str.findall(text_str))        ❽
print(' bytes:', re_words_bytes.findall(text_bytes))   ❾
```

❶ Первые два регулярных выражения типа `str`.
❷ Последние два регулярных выражения типа `bytes`.
❸ Текст Unicode, в котором производится поиск, содержит тамильские цифры числа 1729 (логическая строка продолжается до правой закрывающей скобки).
❹ Эта строка конкатенируется с предыдущей на этапе компиляции (см. раздел 2.4.2 «Конкатенация строковых литералов» справочного руководства (*https://docs.python.org/3/reference/ lexical_analysis.html#string-literal-concatenation*) по языку Python).
❺ Для поиска с помощью регулярного выражения типа `bytes` необходима строка типа `bytes`.
❻ Образец `r'\d+'` типа `str` сопоставляется с тамильскими цифрами и цифрами ASCII.
❼ Образец `rb'\d+'` типа `bytes` сопоставляется только с цифрами ASCII.
❽ Образец `r'\w+'` типа `str` сопоставляется с буквами, надстрочными индексами, тамильскими цифрами и цифрами ASCII.
❾ Образец `rb'\w+'` типа `str` сопоставляется только с ASCII-байтами букв и цифр.

```
1. bash
$ python3 ramanujan.py
Text
  'Ramanujan saw ௧௭௨௯ as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
  str  : ['௧௭௨௯', '1729', '1', '12', '9', '10']
  bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
  str  : ['Ramanujan', 'saw', '௧௭௨௯', 'as', '1729', '1³', '12³', '9³', '10³']
  bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'1', b'12', b'9', b'10']
$
```

Рис. 4.8. Результат выполнения скрипта ramanujan.py из примера 4.23

Это тривиальный пример, призванный подчеркнуть одну мысль: в регулярных выражениях можно употреблять как `str`, так и `bytes`, но во втором случае байты, не принадлежащие диапазону ASCII, не считаются ни цифрами, ни символами, являющимися частью слова.

Для регулярных выражений типа `str` существует флаг `re.ASCII`, при задании которого `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S` производят сопоставление только с байтами ASCII. Подробнее см. документацию по модулю `re` (*https://docs.python.org/3/library/re.html*).

Еще один важный двухрежимный модуль – это `os`.

str и bytes в функциях из модуля os

Ядро GNU/Linux ничего не знает о Unicode, поэтому на практике можно встретить имена файлов, представляющие собой последовательности байтов, которые не являются допустимыми ни в какой разумной кодировке и которые нельзя декодировать в тип `str`. Особенно чувствительны к этой проблеме файловые серверы, имеющие клиентов для разных ОС.

Чтобы обойти эту проблему, все функции из модуля `os`, принимающие имена файлов или пути, могут работать с аргументами типа `str` или `bytes`. Если такая функция вызывается с аргументом типа `str`, то он автоматически преобразуется кодеком, определяемым функцией `sys.getfilesystemencoding()`, а ответ ОС декодируется тем же кодеком. Почти всегда это именно то, что нужно, и согласуется с сэндвичем Unicode.

Но если приходится иметь дело с именами, которые так обрабатывать нельзя (или исправлять такие имена), то можно передавать функциям из модуля `os` аргументы типа `bytes`, и при этом они возвращают значения типа `bytes`. Это позволяет работать с любыми именами файлов и путями, сколько бы в них ни было крокозябров. См. пример 4.24.

Пример 4.24. Функция `listdir` с аргументами типа `str` и `bytes` и ее результаты

```
>>> os.listdir('.') ❶
['abc.txt', 'digits-of-π.txt']
>>> os.listdir(b'.') ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
```

❶ Второе имя файла равно «digits-of-π.txt» (с греческой буквой «пи»).

❷ Если аргумент имеет тип `bytes`, то `listdir` возвращает имена файлов как байты: `b'\xcf\x80'` – представление греческой буквы «пи» в кодировке UTF-8.

Чтобы помочь обрабатывать последовательности типа `str` или `bytes`, составляющие имена файлов или пути, модуль `os` предоставляет специальные функции кодирования и декодирования `os.fsencode(name_or_path)` и `os.fsdecode(name_or_path)`. Обе функции принимают аргумент типа `str` либо `bytes` или, начиная с Python 3.6, объект, реализующий интерфейс `os.PathLike`.

Unicode – глубокая кроличья нора. Пора заканчивать рассказ о типах `str` и `bytes`.

Резюме

Мы начали эту главу с опровержения утверждения `1 символ == 1 байт`. В мире, перешедшем на Unicode, необходимо разделять понятия текстовой строки

и двоичной последовательности, которой такая строка представлена в файле. И Python 3 поддерживает подобное разделение.

После краткого обзора двоичных типов последовательностей – `bytes`, `bytearray` и `memoryview` – мы перешли к кодированию и декодированию, привели репрезентативную выборку кодеков и объяснили, как предотвратить или обработать печально известные ошибки `UnicodeEncodeError`, `UnicodeDecodeError` и `SyntaxError`, вызванные неправильным кодированием исходного файла Python.

Далее мы рассмотрели теорию и практику распознавания кодировки в отсутствие метаданных; теоретически это невозможно, но на практике пакет Chardet неплохо справляется с этой задачей для многих популярных кодировок. Мы сказали о том, что маркеры порядка байтов – единственная информация о кодировке, присутствующая в файлах с кодировкой UTF-16 и UTF-32, иногда также UTF-8.

В следующем разделе мы продемонстрировали открытие текстовых файлов. В этой несложной задаче есть один подвох: именованный аргумент `encoding=` необязателен, хотя должен бы быть таковым. Если кодировка не задана, то получается программа, которая генерирует «простой текст», несовместимый с разными платформами из-за несовпадения кодировок по умолчанию. Затем мы рассказали о различных параметрах, которые интерпретатор Python использует в качестве источников умолчаний: `locale.getpreferredencoding()`, `sys.getfilesystemencoding()`, `sys.getdefaultencoding()`, а также о кодировках стандартных потоков ввода-вывода (например, `sys.stdout.encoding`). Печальным фактом для пользователей Windows является то, что эти параметры зачастую имеют разные значения на одной и той же машине, причем эти значения несовместимы между собой. Напротив, пользователи GNU/Linux и macOS обитают в счастливом мире, где практически повсюду по умолчанию используется кодировка `UTF-8`.

В Unicode некоторые символы можно представить несколькими способами, поэтому перед сравнением необходимо выполнить нормализацию. Мы не только объяснили, что такое нормализация и сворачивание регистра, но и привели несколько служебных функций, которые вы можете приспособить к своим нуждам, и среди них функцию, которая полностью удаляет все акценты. Далее мы видели, как правильно сортировать текст Unicode с применением стандартного модуля `locale` (у которого есть некоторые недостатки) или альтернативного ему внешнего пакета *pyuca*, не зависящего от головоломных настроек локали.

Мы воспользовались базой данных Unicode для написания командной утилиты поиска символов по имени – благодаря мощным средствам Python она уместилась всего в 28 строках кода. Мы кратко рассмотрели другие метаданные Unicode и двухрежимный API, позволяющий вызывать некоторые функции с аргументами типа `str` или `bytes`, получая при этом разные результаты.

Дополнительная литература

Хочу отметить выдающееся выступление Нэда Бэтчелдера на конференции PyCon US 2012 года «Pragmatic Unicode – or – How Do I Stop the Pain?» (*http://nedbatchelder.com/text/unipain.html*). Нэд оказался настолько профессионален, что выложил полную запись доклада со всеми слайдами и видео.

На конференции PyCon 2014 Эстер Нэм и Трэвис Фишер выступили с великолепным докладом «Character encoding and Unicode in Python: How to (╯°□°)╯ ⌢ ┻━┻ with dignity» (слайды имеются по адресу *https://www.slideshare.net/fischertrav/character-encoding-unicode-how-to-with-dignity-33352863*, видео – по адресу *https://pyvideo.org/pycon-us-2014/character-encoding-and-unicode-in-python.html*), из которого я взял эпиграф к данной главе «Человек работает с текстом, компьютер – с байтами».

Леннарт Регебро – один из технических рецензентов книги – представил свою «полезную мысленную модель Unicode» в короткой статье «Unconfusing Unicode: What Is Unicode?» (*https://regebro.wordpress.com/2011/03/23/unconfusing-unicode-what-is-unicode/*). Unicode – сложный стандарт, поэтому мысленная модель Леннарта – действительно полезная отправная точка.

В официальном документе «Unicode HOWTO» (*https://docs.python.org/3/howto/unicode.html*) в документации по Python эта тема рассматривается с разных точек зрения: от удачного исторического введения до деталей синтаксиса, кодеков, регулярных выражений, имен файлов и рекомендаций по написанию кода ввода-вывода с учетом Unicode (сэндвич Unicode). В каждом разделе имеются ссылки на дополнительную информацию. В главе 4 «Строки» (*http://www.diveintopython3.net/strings.html*) замечательной книги Mark Pilgrim «Dive into Python 3» (*http://www.diveintopython3.net*) также имеется отличное введение в поддержку Unicode в Python 3. В главе 15 той же книги (*https://finderiko.com/python-book*) описан перенос библиотеки Chardet с Python 2 на Python 3 – ценный пример, учитывая, что переход от старого типа `str` к новому типу `bytes` стал причиной большинства неприятностей, связанных с миграцией, а это – как раз основная тема библиотеки, призванной распознавать кодировки.

Для тех, кто знает Python 2, но незнаком с Python 3, в статье Гвидо ван Россума «What's New in Python 3.0» (*https://docs.python.org/3.0/whatsnew/3.0.html#text-vs-data-instead-of-unicode-vs-8-bit*) перечислено 15 основных отличий со множеством ссылок. Гвидо начинает с прямого заявления: «Все, что, как вам казалось, вы знали о двоичных данных и Unicode, изменилось». Армен Ронашер (Armin Ronacher) опубликовал в своем блоге статью «The Updated Guide to Unicode on Python» (*https://lucumr.pocoo.org/2013/7/2/the-updated-guide-to-unicode/*), в которой акцентирует внимание на некоторых подводных камнях Unicode в Python 3 (Армен – не большой поклонник Python 3).

В главе 2 книги David Beazley, Brian K. Jones «*Python Cookbook*», 3-е издание (O'Reilly), приведено несколько рецептов, относящихся к нормализации в Unicode, очистке текста и выполнению текстовых операций над последовательностями байтов. В главе 5, посвященной файлам и вводу-выводу, имеется рецепт 5.17 «Запись байтов в текстовый файл», где показано, что за любым текстовым файлом стоит двоичный поток, к которому при желании можно получить доступ напрямую. Далее в рецепте 6.11 «Чтение и запись двоичных массивов структур» показано применение модуля `struct`.

В блоге Ника Кофлина (Nick Coghlan) «Python Notes» есть две статьи, имеющие непосредственное отношение к этой главе: «Python 3 and ASCII Compatible Binary Protocols» (*http://python-notes.curiousefficiency.org/en/latest/*) и «Processing Text Files in Python 3» (*http://python-notes.curiousefficiency.org/en/latest/python3/text_file_processing.html*). Настоятельно рекомендую.

Список кодировок, поддерживаемых Python, приведен в разделе «Стандартные кодировки» (*https://docs.python.org/3/library/codecs.html#standard-encodings*) документации по модулю `codecs`. О том, как получить доступ к этому списку из программы, см. скрипт */Tools/unicode/listcodecs.py* (*https://github.com/python/cpython/blob/0bbf30e2b910bc9c5899134ae9d73a8df968da35/Tools/unicode/listcodecs.py*), входящий в состав дистрибутива CPython.

Книги Jukka K. Korpela «Unicode Explained» (*https://www.oreilly.com/library/view/unicode-explained/059610121X/*) (O'Reilly) и Richard Gillam «Unicode Demystified» (Addison-Wesley) не связаны с Python, но очень помогли мне в изучении концепций Unicode. Книга Виктора Стиннера (Victor Stinner) «Programming with Unicode» (*http://unicodebook.readthedocs.org/index.html*) – бесплатное произведение, опубликованное самим автором (распространяется по лицензии Creative Commons BY-SA); в ней рассматривается как сам стандарт Unicode, так и инструментальные средства и API для основных операционных систем и нескольких языков программирования, включая Python.

На страницах сайта W3C «Case Folding: An Introduction» (*http://www.w3.org/International/wiki/Case_folding*) и «Character Model for the World Wide Web: String Matching and Searching» (*http://www.w3.org/TR/charmod-norm/*) рассматривается концепция нормализации; первый документ написан в форме введения для начинающих, а второй – рабочий проект, изложенный сухим языком стандарта – в том же стиле, что «Unicode Standard Annex #15 – Unicode Normalization Forms» (*http://unicode.org/reports/tr15/*). Документ «Frequently Asked Questions / Normalization» (*http://www.unicode.org/faq/normalization.html*) на сайте Unicode.org (*http://www.unicode.org/*) проще для восприятия, равно как и NFC FAQ (*http://www.macchiato.com/unicode/nfc-faq*) Марка Дэвиса – автора нескольких алгоритмов Unicode и президента консорциума Unicode Consortium на момент работы над этой книгой.

В 2016 году Музей современного искусства в Нью-Йорке добавил в свою коллекцию оригинальные эмодзи (*https://stories.moma.org/the-original-emoji-set-has-been-added-to-the-museum-of-modern-arts-collection-c6060e141f61*) – 176 эмодзи, разработанных дизайнером Шигетака Курита в 1999 году для NTT DOCOMO, японского мобильного оператора. Углубляясь в историю еще дальше, отметим, что *Эмодзипедиа* (*https://blog.emojipedia.org/*) опубликовала статью «Поправки к истории первого набора эмодзи» (*https://blog.emojipedia.org/correcting-the-record-on-the-first-emoji-set/*), в которой честь создания самого раннего из известных наборов эмодзи приписывается японскому банку SoftBank, который разместил его в сотовых телефонах в 1997 году. Набор SoftBank является источником 90 эмодзи, находящихся ныне в Unicode, включая U+1F4A9 (`PILE OF POO`, `КАКАШКА`). Сайт Мэтью Ротенберга *emojitracker.com* – динамическая панель, на которой в режиме реального времени показываются счетчики использования эмодзи в Твиттере. Когда я пишу эти слова, самым популярным был эмодзи `FACE WITH TEARS OF JOY` (`ЛИЦО СО СЛЕЗАМИ РАДОСТИ`) (U+1F602), он встречался 3 313 667 315 раз.

Поговорим

Не-ASCII имена в исходном коде: стоит ли их использовать?

В Python 3 разрешается использовать в исходном коде идентификаторы в кодировке, отличной от ASCII:

```
>>> ação = 'PBR' # ação = frwbz
>>> ε = 10**-6 # ε = epsilon
```

Некоторым эта идея не нравится. Чаще всего в пользу использования только ASCII-идентификаторов приводят тот довод, что так всем будет проще читать и редактировать код. Но тут упущена одна деталь: вы хотите, чтобы ваш исходный код был доступен для чтения и редактирования целевой аудитории, а это вовсе необязательно «все». Если код принадлежит интернациональной корпорации или является открытым, и вы хотите, чтобы дополнять его могли люди во всего света, то идентификаторы определенно стоит писать по-английски, и тогда вам нужна только кодировка ASCII.

Но если вы преподаете в Бразилии, то студентам будет проще читать код, в котором имена переменных и функций написаны по-португальски и притом без орфографических ошибок. И у них не будет проблем с вводом седилей и акцентированных гласных, поскольку они работают с местной клавиатурой.

Поскольку теперь Python понимает Unicode-имена, а UTF-8 – кодировка исходного кода по умолчанию, то я не вижу никакого смысла писать португальские идентификаторы без диакритических знаков, как мы делали в Python 2 по необходимости, – если, конечно, вы не собираетесь запускать свой код и в Python 2 тоже. Если имена все равно португальские, то отбрасывание диакритических знаков не сделает код понятнее ни для кого.

Это моя личная точка зрения – человека, говорящего на бразильском диалекте португальского языка, – но полагаю, что она применима и к другим культурам и регионам: выберите естественный язык, который наиболее понятен потенциальным читателям кода, и набирайте его символы правильно.

Что такое «простой текст»?

Для любого, кто в повседневной работе имеет дело с неанглоязычными текстами, «простой текст» не ассоциируется с «ASCII». В глоссарии Unicode (*http://www.unicode.org/glossary/#plain_text*) *простой текст* определяется так:

> Закодированный для компьютера текст, который включает только последовательность кодовых позиций из некоторого стандарта – без какой-либо форматной или структурной информации.

Начинается это определение очень хорошо, но с частью после тире я не согласен. HTML – прекрасный пример простого текста, содержащего форматную и структурную информацию. Но это все же простой текст, потому что каждый байт в таком файле представляет некий текстовый символ, обычно в кодировке UTF-8. В файле нет байтов, несущих нетекстовую нагрузку, как, скажем, в файлах типа PNG или XLS, где большинство байтов – это упакованные двоичные значения, представляющие либо цвета в формате RGB, либо числа с плавающей точкой. В простом тексте число было бы представлено в виде последовательности цифр.

Я пишу эту книгу в формате простого текста, который, по иронии судьбы, называется AsciiDoc (*http://www.methods.co.nz/asciidoc/*) и является частью великолепного инструментария, входящего в комплект платформы книгоиздания Atlas компании O'Reilly (*https://atlas.oreilly.com/*). Исходные файлы в формате AsciiDoc – это простой текст, но в кодировке UTF-8, а не ASCII. В противном случае писать эту главу было бы крайне затруднительно. Несмотря на свое название, AsciiDoc – отличная вещь.

Вселенная Unicode постоянно расширяется, и на ее границах не всегда есть подходящие инструменты. Не все нужные мне символы присутствовали в шрифтах, которыми набрана эта книга. Именно поэтому я был вынужден использовать изображения, а не листинги в некоторых примерах из этой главы. С другой стороны, на терминалах в Ubuntu и macOS большинство символов Unicode прекрасно отображаются – включая и японские символы, составляющие слово «mojibake»: 文字化け.

Как кодовые позиции представлены в памяти?

В официальном руководстве по Python старательно обходится вопрос о том, как кодовые позиции строки `str` хранятся в памяти. В конце концов, это действительно деталь реализации. Теоретически это не имеет значения: каким бы ни было внутреннее представление, каждая строка при выводе должна перекодироваться в объект типа `bytes`.

В Python 3 объект `str` хранится в памяти как последовательность кодовых позиций с фиксированным количеством байтов на одну позицию, чтобы обеспечить прямой доступ к любому символу или срезу.

Начиная с версии Python 3.3 интерпретатор, создавая объект `str`, проверяет, из каких символов он состоит, и выбирает наиболее экономичное размещение в памяти данного объекта. Если имеются только символы из диапазона `latin1`, то каждая кодовая позиция `str` будет представлена всего одним байтом. В противном случае для представления кодовой позиции может понадобиться 2 или 4 байта – все зависит от `str`. Это упрощенное изложение, детали см. в документе PEP 393 «Flexible String Representation» (*https://www.python.org/dev/peps/pep-0393/*).

Гибкое представление строки похоже на представление типа `int` в Python 3: если целое число умещается в машинном слове, то оно и хранится как одно машинное слово. В противном случае интерпретатор переходит на представление переменной длины, как для типа `long` в Python 2. Приятно видеть, как распространяются хорошие идеи.

Однако мы всегда можем рассчитывать на Армена Ронахера, когда нужно найти проблемы в Python 3. Он объяснил мне, почему на практике эта идея не так уж хороша: берется единственный символ `RAT` (U+1F400), в результате чего текст, который без этого состоял бы только из символов ASCII, разбухает и превращается в массив – пожиратель памяти, в котором на каждый символ отведено четыре байта, тогда как для всех символов, кроме `RAT`, хватило бы и одного. Кроме того, из-за многочисленных способов комбинирования символов в Unicode умение быстро извлекать произвольный символ по номеру позиции переоценено, а попытка извлекать произвольные участки из Unicode-текста в лучшем случае наивна, а зачастую и вредна, поскольку порождает крокозябры. По мере распространения эмодзи эти проблемы будут только нарастать.

Глава 5

Построители классов данных

> Классы данных – как дети. В начале пути они прелестны, но чтобы участвовать во взрослой жизни, должны брать на себя ответственность.
>
> – Мартин Фаулер и Кент Бек[1]

Python предлагает несколько способов построить простой класс, являющийся просто набором полей, почти или даже совсем без дополнительной функциональности. Этот паттерн называется «классом данных», а `dataclasses` – один из пакетов, поддерживающих его. В этой главе рассматриваются три разных построителя классов, которые можно использовать, чтобы сократить время написания классов данных.

`collections.namedtuple`

Самый простой способ, доступен начиная с версии Python 2.6.

`typing.NamedTuple`

Альтернатива, требующая заданий аннотаций типов для полей. Существует начиная с версии Python 3.5, а синтаксическая конструкция `class` была добавлена в версии 3.6.

`@dataclasses.dataclass`

Декоратор класса, допускающий бóльшую гибкость настройки, чем предыдущие альтернативы. Но вместе с богатством параметров приходит и дополнительная сложность. Существует начиная с версии Python 3.7.

Рассмотрев все эти построители, мы обсудим, почему «класс данных» считается запашком в коде, т. е. паттерном, который может быть симптомом неудачного объектно-ориентированного дизайна.

Может показаться, что `typing.TypedDict` – еще один построитель классов данных. В нем используется похожий синтаксис, да и в документации по модулю `typing` для Python 3.9 его описание идет сразу после `typing.NamedTuple`.

Однако `TypedDict` не строит конкретные классы, для которых можно было бы создать экземпляры. Это всего лишь синтаксис для написания аннотаций типов параметров функций и переменных. Такие аннотации принимают отображения, используемые в качестве записей, ключами которых являются имена полей. Мы будем рассматривать их в главе 15 «TypedDict».

[1] Из книги «Refactoring», первое издание, глава 3, раздел «Bad Smells in Code, Data Class», стр. 87 (Addison-Wesley).

Что нового в этой главе

В первом издании этой главы вообще не было. Раздел «Классические именованные кортежи» входил в главу 2, но все остальное написано заново.

Мы начнем с общего обзора всех трех построителей классов.

Обзор построителей классов данных

Рассмотрим простой класс для представления географических координат (пример 5.1).

Пример 5.1. class/coordinates.py

```
class Coordinate:

    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon
```

Класс `Coordinate` хранит в своих атрибутах широту и долготу. Написание стереотипного кода `__init__` очень быстро надоедает, особенно если в классе атрибутов не два, а больше: каждый придется упомянуть три раза! И к тому же такой стереотипный код не дает нам того, чего мы ожидаем от объекта Python:

```
>>> from coordinates import Coordinate
>>> moscow = Coordinate(55.76, 37.62)
>>> moscow
<coordinates.Coordinate object at 0x107142f10> ❶
>>> location = Coordinate(55.76, 37.62)
>>> location == moscow ❷
False
>>> (location.lat, location.lon) == (moscow.lat, moscow.lon) ❸
True
```

❶ Метод `__repr__`, унаследованный от `object`, не особенно полезен.

❷ Оператор `==` вообще бессмысленный; метод `__repr__`, унаследованный от `object`, сравнивает идентификаторы объектов.

❸ Для сравнения двух координат необходимо явно сравнить обе пары атрибутов.

Построители классов данных, рассматриваемые в этой главе, автоматически предоставляют необходимые методы `__init__`, `__repr__` и `__eq__`, а также другие полезные средства.

Ни один из обсуждаемых ниже построителей классов не опирается на наследование. Построители `collections.namedtuple` и `typing.NamedTuple` строят классы, являющиеся подклассами `tuple`. А `@dataclass` – вообще декоратор класса, который никак не влияет на иерархию классов. Все они пользуются различными средствами метапрограммирования для внедрения методов и атрибутов данных в создаваемый класс.

Ниже показано поведение класса `Coordinate`, построенного с помощью `namedtuple` – фабричной функции, которая строит подкласс `tuple` с заданными нами именем и полями:

```
>>> from collections import namedtuple
>>> Coordinate = namedtuple('Coordinate', 'lat lon')
>>> issubclass(Coordinate, tuple)
True
>>> moscow = Coordinate(55.756, 37.617)
>>> moscow
Coordinate(lat=55.756, lon=37.617) ❶
>>> moscow == Coordinate(lat=55.756, lon=37.617) ❷
True
```

❶ Полезный метод `__repr__`.
❷ Осмысленный метод `__eq__`.

Появившийся позже класс `typing.NamedTuple` предлагает такую же функциональность и дополнительно аннотации типа для каждого поля:

```
>>> import typing
>>> Coordinate = typing.NamedTuple('Coordinate',
...     [('lat', float), ('lon', float)])
>>> issubclass(Coordinate, tuple)
True
>>> typing.get_type_hints(Coordinate)
{'lat': <class 'float'>, 'lon': <class 'float'>}
```

Типизированный именованный кортеж можно построить так же, задав поля в виде именованных аргументов:

```
Coordinate = typing.NamedTuple('Coordinate', lat=float, lon=float)
```

Этот код проще читать, а кроме того, он позволяет задать отображение полей на типы в виде `**fields_and_types`.

Начиная с версии Python 3.6 `typing.NamedTuple` можно также использовать в предложении `class`, а аннотации типов записывать, как предлагается в документе PEP 526 «Syntax for Variable Annotations» (*https://peps.python.org/pep-0526/*). Это гораздо проще читается и позволяет легко переопределять методы или добавлять новые. В примере 5.2 показан тот же класс `Coordinate` с двумя атрибутами `float` и методом `__str__`, который отображает координаты в формате 55.8°N, 37.6°E.

Пример 5.2. typing_namedtuple/coordinates.py

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float
    lon: float

    def __str__(self):
        ns = 'N' if self.lat >= 0 else 'S'
        we = 'E' if self.lon >= 0 else 'W'
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°{we}'
```

Хотя `NamedTuple` выглядит в предложении `class` как суперкласс, на самом деле это не так. В классе `typing.NamedTuple` используется продвинутая функциональность метакласса[1], позволяющая настроить создание пользовательского класса. В этом легко убедиться:

```
>>> issubclass(Coordinate, typing.NamedTuple)
False
>>> issubclass(Coordinate, tuple)
True
```

В методе `__init__`, сгенерированном с помощью `typing.NamedTuple`, поля встречаются в том же порядке, в каком они перечислены в предложении `class`.

Как и `typing.NamedTuple`, декоратор `dataclass` поддерживает синтаксис объявления атрибутов экземпляра, описанный в документе PEP 526. Декоратор читает аннотации переменных и автоматически генерирует методы вашего класса. Для сравнения взгляните на эквивалентный класс `Coordinate`, написанный с применением декоратора `dataclass` (пример 5.3).

Пример 5.3. dataclass/coordinates.py

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Coordinate:
    lat: float
    lon: float

    def __str__(self):
        ns = 'N' if self.lat >= 0 else 'S'
        we = 'E' if self.lon >= 0 else 'W'
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°{we}'
```

Отметим, что тела классов в примерах 5.2 и 5.3 одинаковы – разница только в самом предложении `class`. Декоратор `@dataclass` не опирается ни на наследование, ни на метакласс, поэтому не препятствует любому использованию вами этих механизмов[2]. Класс `Coordinate` в примере 5.3 – подкласс `object`.

Основные возможности

У различных построителей классов данных много общих черт, сведенных в табл. 5.1.

Таблица 5.1. Сравнение избранных возможностей всех трех построителей классов данных (`x` означает экземпляр класса данных)

	namedtuple	NamedTuple	dataclass
Изменяемые экземпляры	Нет	Нет	Да
Синтаксис предложения `class`	Нет	Да	Да

[1] Метаклассы – одна из тем главы 24 «Метапрограммирование классов».

[2] Декораторы классов рассматриваются в главе 24 «Метапрограммирование классов» наряду с метаклассами. То и другое – способы настройки поведения классов, недостижимые с помощью наследования.

Окончания табл. 5.1

	namedtuple	NamedTuple	dataclass
Конструирование dict	`x._asdict()`	`x._asdict()`	`dataclasses.asdict(x)`
Получение имен полей	`x._fields`	`x._fields`	`[f.name for f in dataclasses.fields(x)]`
Получение значений по умолчанию	`x._field_defaults`	`x._field_defaults`	`[f.default for f in dataclasses.fields(x)]`
Получение типов полей	Не применимо	`x.__annotations__`	`x.__annotations__`
Новый экземпляр с изменениями	`x._replace(…)`	`x._replace(…)`	`dataclasses.replace(x, …)`
Новый класс во время выполнения	`namedtuple(…)`	`NamedTuple(…)`	`dataclasses.make_dataclass(…)`

У классов, построенных с помощью `typing.NamedTuple` и `@dataclass`, имеется атрибут `__annotations__`, в котором хранятся аннотации типов полей. Однако читать `__annotations__` напрямую не рекомендуется. Лучше получать эту информацию от метода `inspect.get_annotations(MyClass)` (добавлен в Python 3.10) или `typing.get_type_hints(MyClass)` (Python 3.5–3.9). Дело в том, что эти функции предоставляют дополнительные возможности, в частности разрешение опережающих ссылок в аннотациях типов. Мы вернемся к этому вопросу гораздо позже, в разделе «Проблемы аннотаций на этапе выполнения» главы 15.

А теперь обсудим эти основные возможности.

Изменяемые экземпляры

Ключевое различие между этими построителями классов – тот факт, что `collections.namedtuple` и `typing.NamedTuple` строят подклассы `tuple`, поэтому экземпляры оказываются неизменяемыми. По умолчанию `@dataclass` порождает изменяемые классы. Но декоратор принимает именованный аргумент `frozen`, показанный в примере 5.3. Если `frozen=True`, то класс возбудит исключение при попытке присвоить какому-либо полю значение после инициализации экземпляра.

Синтаксис предложения class

Только `typing.NamedTuple` и `dataclass` поддерживают регулярный синтаксис предложения `class`, что упрощает добавление методов и строк документации в создаваемый класс.

Конструирование dict

Оба варианта на основе именованных кортежей предоставляют метод экземпляра (`._asdict`) для конструирования объекта `dict` по полям экземпляра класса данных. Модуль `dataclasses` содержит функцию для этой цели: `dataclasses.asdict`.

Получение имен полей и значений по умолчанию

Все три построителя классов позволяют получать имена полей и заданных для них значений по умолчанию. В классах, производных от именованного кортежа, эти метаданные хранятся в атрибутах `._fields` и `._fields_defaults`. От класса с декоратором `dataclass` те же метаданные можно получить с помощью функции `fields` из модуля `dataclasses`. Она возвращает кортеж объектов `Field`, имеющих несколько атрибутов, в т. ч. `name` и `default`.

Получение типов полей

В классах, определенных с помощью `typing.NamedTuple` и `@dataclass`, имеется атрибут `__annotations__`, содержащий отображение имен полей на их типы. Как уже было сказано, рекомендуется использовать функцию `typing.get_type_hints`, а не читать этот атрибут непосредственно.

Новый экземпляр с изменениями

Если дан именованный кортеж `x`, то вызов `x._replace(**kwargs)` возвращает новый экземпляр, в котором значения некоторых атрибутов изменены в соответствии с переданными именованными аргументами. Функция уровня модуля `dataclasses.replace(x, **kwargs)` делает то же самое для экземпляра класса с декоратором `dataclass`.

Новый класс во время выполнения

Хотя синтаксис предложения `class` более понятен, он предполагает статическое создание. Иногда требуется создавать классы данных динамически, во время выполнения. Для этого можно использовать синтаксис вызова функции `collections.namedtuple`, поддерживаемый также классом `typing.NamedTuple`. В модуле `dataclasses` для той же цели имеется функция `make_dataclass`.

После этого обзора основных возможностей построителей классов данных рассмотрим каждый из них подробно и начнем с простейшего.

Классические именованные кортежи

Функция `collections.namedtuple` – это фабрика, порождающая подклассы `tuple`, дополненные возможностью задавать имена полей, имя класса и информативный метод `__repr__`. Классы, построенные с помощью `namedtuple`, можно использовать всюду, где нужны кортежи, и на самом деле многие функции из стандартной библиотеки Python, которые служат для возврата кортежей, теперь для удобства возвращают именованные кортежи; на пользовательский код это никак не влияет.

Экземпляры класса, построенного с помощью `namedtuple`, потребляют ровно столько памяти, сколько кортежи, потому что имена полей хранятся в определении класса.

В примере 5.4 показано, как можно было бы определить кортеж для хранения информации о городе.

Пример 5.4. Определение и использование именованного кортежа

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population coordinates') ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667)) ❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722, 139.691667))
>>> tokyo.population ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

❶ Для создания именованного кортежа нужно задать два параметра: имя класса и список имен полей; последний может быть любым итерируемым объектом, содержащим строки, или одной строкой, в которой имена перечислены через запятую.
❷ Данные передаются конструктору в виде позиционных аргументов (тогда как конструктор кортежа принимает единственный итерируемый объект).
❸ К полям можно обращаться по имени или по номеру позиции.

Будучи подклассом `tuple`, `City` наследует полезные методы, в частности `__eq__` и специальные методы, поддерживающие операторы сравнения, включая `__lt__`, что позволяет сортировать списки экземпляров `City`.

Именованный кортеж предлагает несколько атрибутов и методов в дополнение к унаследованным от кортежа. В примере 5.5 показаны наиболее полезные: атрибут класса `_fields`, метод класса `_make(iterable)` и метод экземпляра `_asdict()`.

Пример 5.5. Атрибуты и методы именованного кортежа (продолжение предыдущего примера)

```
>>> City._fields ❶
('name', 'country', 'population', 'location')
>>> Coordinate = namedtuple('Coordinate', 'lat lon')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, Coordinate(28.613889, 77.208889))
>>> delhi = City._make(delhi_data) ❷
>>> delhi._asdict() ❸
{'name': 'Delhi NCR', 'country': 'IN', 'population': 21.935,
'location': Coordinate(lat=28.613889, lon=77.208889)}
>>> import json
>>> json.dumps(delhi._asdict()) ❹
'{"name": "Delhi NCR", "country": "IN", "population": 21.935,
"location": LatLong(lat=28.613889, long=77.208889)
>>>
```

❶ `_fields` – кортеж, содержащий имена полей данного класса.
❷ `_make()` строит объект `City` из итерируемого объекта; конструктор `City(*delhi_data)` делает то же самое.
❸ `._asdict()` возвращает объект `dict`, построенный по именованному кортежу.
❹ `._asdict()` полезен для сериализации данных в формате JSON.

До версии Python 3.7 включительно метод `_asdict` возвращал `OrderedDict`. Начиная с версии Python 3.8 он возвращает простой `dict`, и это хорошо, потому что теперь мы знаем, что порядок вставки ключей сохраняется. Если вам обязательно нужен `OrderedDict`, то документация по `_asdict` рекомендует надстроить его над результатом: `OrderedDict(x._asdict())`.

Начиная с версии Python 3.7 `namedtuple` принимает чисто именованный аргумент `defaults`, предоставляющий итерируемый объект, содержащий *N* значений по умолчанию для каждого из *N* самых правых полей класса. В примере 5.6 показано, как определить именованный кортеж `Coordinate` со значением по умолчанию для поля `reference`.

Пример 5.6. Атрибуты и методы именованного кортежа (продолжение примера 5.5)

```
>>> Coordinate = namedtuple('Coordinate', 'lat lon reference', defaults=['WGS84'])
>>> Coordinate(0, 0)
Coordinate(lat=0, lon=0, reference='WGS84')
>>> Coordinate._field_defaults
{'reference': 'WGS84'}
```

В абзаце «Синтаксис предложения class» выше я упомянул, что проще кодировать методы, используя синтаксис класса, который поддерживается классом `typing.NamedTuple` и декоратором `@dataclass`. В класс, созданный с помощью `namedtuple`, тоже можно добавить методы, но это хак. Если хаки вас не интересуют, можете пропустить следующую врезку.

Вставка метода в класс, созданный с помощью namedtuple

Вспомните, как мы строили класс `Card` в примере 1.1 главы 1:

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

Затем я написал функцию `spades_high` для сортировки. Было бы хорошо инкапсулировать ее логику в методе класса `Card`, но чтобы добавить `spades_high` в `Card`, не пользуясь возможностями предложения `class`, нужно прибегнуть к хаку: определить функцию, а затем присвоить ее атрибуту класса. В примере 5.7 показано, как это делается.

Пример 5.7. frenchdeck.doctest: добавление атрибута класса и метода в Card, именованный кортеж из примера «Колода карт на Python»

```
>>> Card.suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0) ❶
>>> def spades_high(card): ❷
...     rank_value = FrenchDeck.ranks.index(card.rank)
...     suit_value = card.suit_values[card.suit]
...     return rank_value * len(card.suit_values) + suit_value
...
>>> Card.overall_rank = spades_high ❸
>>> lowest_card = Card('2', 'clubs')
>>> highest_card = Card('A', 'spades')
>>> lowest_card.overall_rank() ❹
```

```
0
>>> highest_card.overall_rank()
51
```

❶ Присоединить атрибут класса, содержащий достоинства для каждой масти.

❷ Функция `spades_high` станет методом; первый аргумент необязательно называть `self`. Ему все равно будет присвоена ссылка на объект, от имени которого вызывался метод.

❸ Присоединить функцию к классу `Card` как метод с именем `overall_rank`.

❹ Работает!

Для удобочитаемости и удобства сопровождения в будущем гораздо лучше помещать методы внутрь предложения `class`. Но и об этом приеме полезно знать – может пригодиться[1].

Это было небольшое отступление для демонстрации мощи динамического языка.

Теперь перейдем к рассмотрению варианта `typing.NamedTuple`.

Типизированные именованные кортежи

Класс `Coordinate` с полем, имеющим значение по умолчанию, из примера 5.6 можно написать, воспользовавшись классом `typing.NamedTuple`, как показано в примере 5.8.

Пример 5.8. typing_namedtuple/coordinates2.py

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float ❶
    lon: float
    reference: str ='WGS84' ❷
```

❶ Каждое поле экземпляра должно быть аннотировано типом.

❷ Поле экземпляра `reference` аннотировано типом и значением по умолчанию.

Классы, построенные с помощью `typing.NamedTuple`, не имеют методов, помимо тех, что генерирует `collections.namedtuple`, а также унаследованных от `tuple`. Единственное различие – присутствие атрибута класса `__annotations__`, который интерпретатор Python игнорирует во время выполнения.

Поскольку основной отличительной особенностью `typing.NamedTuple` являются аннотации типов, кратко рассмотрим их, прежде чем вернуться к изучению построителей классов данных.

[1] Если вы знакомы с Ruby, то знаете, что внедрение методов – хорошо известный прием среди рубистов, однако отношение к нему неоднозначное. В Python он не так широко распространен, поскольку неприменим к встроенным типам: `str`, `list` и т. д. Я считаю это ограничение Python благословением.

Краткое введение в аннотации типов

Аннотации типов – это способ объявить ожидаемые типы аргументов и возвращаемого значения функции, переменных и атрибутов.

Первое, что нужно знать об аннотациях типов, – то, что ни компилятор байт-кода, ни интерпретатор Python их никак не контролируют.

Это очень краткое введение в аннотации типов, его единственная цель – чтобы вы поняли синтаксис и семантику аннотаций, используемых в объявлениях с применением `typing.NamedTuple` и `@dataclass`. Мы будем рассматривать аннотации типов для сигнатур функций в главе 8, а более сложные аннотации в главе 15. Здесь же будут показаны в основном аннотации, в которых участвуют простые встроенные типы, например `str`, `int` и `float`, но, откровенно говоря, именно ими чаще всего аннотируют поля классов данных.

Никаких последствий во время выполнения

Аннотации типов в Python правильнее всего рассматривать как «документацию, которая может быть проверена IDE и программами проверки типов».

Дело в том, что аннотации типов не оказывают никакого влияния на поведение Python-программ во время выполнения. Взгляните на пример 5.9.

Пример 5.9. Python не проверяет аннотации типов во время выполнения

```
>>> import typing
>>> class Coordinate(typing.NamedTuple):
...     lat: float
...     lon: float
...
>>> trash = Coordinate('Ni!', None)
>>> print(trash)
Coordinate(lat='Ni!', lon=None) ❶
```

❶ Я предупреждал: никаких проверок во время выполнения!

Если вы оформите код из примера 5.9 в виде модуля Python, то при выполнении он выведет бессмысленные координаты, не выдав ни ошибки, ни предупреждения:

```
$ python3 nocheck_demo.py
Coordinate(lat='Ni!', lon=None)
```

Аннотации типов предназначены прежде всего для сторонних программ проверки типов, например Муру (*https://mypy.readthedocs.io/en/stable/*) или интегрированной среды разработки PyCharm (*https://www.jetbrains.com/pycharm/*) со встроенной проверкой типов. Это инструменты статического анализа: они проверяют «покоящийся» исходный код, а не код в процессе выполнения.

Чтобы увидеть эффект от аннотаций типов, нужно применить какой-нибудь из этих инструментов к своему коду. Например, вот что говорит Муру о предыдущем примере:

```
$ mypy nocheck_demo.py
nocheck_demo.py:8: error: Argument 1 to "Coordinate" has
incompatible type "str"; expected "float"
nocheck_demo.py:8: error: Argument 2 to "Coordinate" has
incompatible type "None"; expected "float"
```

Видя определение `Coordinate`, Муру понимает, что оба аргумента конструктора должны иметь тип `float`, но при создании `trash` заданы аргументы типа `str` и `None`[1].

Теперь поговорим о синтаксисе и семантике аннотаций типов.

Синтаксис аннотаций переменных

И `typing.NamedTuple`, и `@dataclass` пользуются синтаксисом аннотаций переменных, определенным в документе PEP 526 (*https://peps.python.org/pep-0526/*). Ниже приведено краткое введение в этот синтаксис в контексте определения атрибутов в предложениях `class`.

Базовый синтаксис аннотации переменной имеет вид:

```
var_name: some_type
```

В разделе «Допустимые аннотации типов» документа PEP 484 (*https://peps.python.org/pep-0484/#acceptable-type-hints*) объясняется, что такое допустимые типы, но в контексте определения класса данных наиболее полезными, скорее всего, будут следующие типы:

- конкретный класс, например `str` или `FrenchDeck`;
- параметризованный тип коллекции, например `list[int]`, `tuple[str, float]` и т. д.;
- `typing.Optional`, например `Optional[str]`, для объявления поля, которое может принимать значения типа `str` или `None`.

Переменную можно также инициализировать значением. В объявлении с помощью `typing.NamedTuple` или `@dataclass` это значение станет значением соответствующего атрибута по умолчанию, если для него не задан аргумент при вызове конструктора:

```
var_name: some_type = a_value
```

Семантика аннотаций переменных

В разделе «Никаких последствий во время выполнения» мы видели, что аннотации типов никак не влияют на происходящее на этапе выполнения. Но на этапе импорта, когда модель загружается, Python читает их и строит словарь `__annotations__`, который `typing.NamedTuple` и `@dataclass` затем используют для дополнения класса.

Начнем это исследование с простого класса в примере 5.10, чтобы впоследствии можно было увидеть, что добавляют `typing.NamedTuple` и `@dataclass`.

---

[1] В контексте аннотаций типов `None` – не синглтон типа `NoneType`, а псевдоним самого типа `NoneType`. Если задуматься, то это выглядит странно, но согласуется с нашей интуицией и заставляет функцию возвращать аннотации, которые проще читать в часто встречающемся случае функций, возвращающих `None`.

Пример 5.10. meaning/demo_plain.py: простой класс с аннотациями типов

```
class DemoPlainClass:
    a: int          ❶
    b: float = 1.1  ❷
    c = 'spam'      ❸
```

❶ Для поля `a` заводится запись в `__annotations__`, но больше оно никак не используется: атрибут с именем `a` в классе не создается.
❷ Поле `b` сохраняется в аннотациях и, кроме того, становится атрибутом класса со значением `1.1`.
❸ `c` – это самый обычный атрибут класса, а не аннотация.

Все это можно проверить на консоли: сначала прочитаем атрибут `__annotations__` класса `DemoPlainClass`, а затем попытаемся получить его атрибуты `a`, `b` и `c`:

```
>>> from demo_plain import DemoPlainClass
>>> DemoPlainClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoPlainClass.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'DemoPlainClass' has no attribute 'a'
>>> DemoPlainClass.b
1.1
>>> DemoPlainClass.c
'spam'
```

Отметим, что интерпретатор создает специальный атрибут `__annotations__`, чтобы запомнить аннотации типов, встречающиеся в исходном коде, – даже в обычном классе.

От поля `a` остается только аннотация. Оно не становится атрибутом класса, потому что с ним не связано никакое значение[1]. Поля `b` и `c` сохраняются в качестве атрибутов класса, потому что с ними связаны значения.

Ни один из этих трех атрибутов не присутствует в новом экземпляре класса `DemoPlainClass`. Если создать объект `o = DemoPlainClass()`, то `o.a` возбуждает исключение `AttributeError`, тогда как `o.b` и `o.c` извлекают атрибуты класса со значениями `1.1` и `'spam'` – обычное поведение объекта в Python.

Инспекция typing.NamedTuple

Теперь исследуем класс, построенный с помощью `typing.NamedTuple` (пример 5.11) с теми же атрибутами и аннотациями, что в `DemoPlainClass` из примера 5.10.

Пример 5.11. meaning/demo_nt.py: класс, построенный с помощью `typing.NamedTuple`

```
import typing

class DemoNTClass(typing.NamedTuple):
    a: int          ❶
    b: float = 1.1  ❷
    c = 'spam'      ❸
```

[1] В Python нет понятия *undefined*, одной из самых глупых ошибок, допущенных при проектировании JavaScript. Спасибо, Гвидо!

❶ Для поля `a` заводится аннотация и атрибут экземпляра.
❷ Для поля `b` также заводится аннотация, и оно также становится атрибутом экземпляра со значением по умолчанию `1.1`.
❸ `c` – самый обычный атрибут класса, никакая аннотация на него не ссылается.

Инспекция `DemoNTClass` дает такие результаты:

```
>>> from demo_nt import DemoNTClass
>>> DemoNTClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoNTClass.a
<_collections._tuplegetter object at 0x101f0f940>
>>> DemoNTClass.b
<_collections._tuplegetter object at 0x101f0f8b0>
>>> DemoNTClass.c
'spam'
```

Здесь для `a` и `b` аннотации такие же, как в примере 5.10. Но `typing.NamedTuple` создает атрибуты класса `a` и `b`. Атрибут `c` – обычный атрибут класса со значением `'spam'`.

Атрибуты класса `a` и `b` являются *дескрипторами*; этот продвинутый механизм рассматривается в главе 23. Пока считайте, что это некий аналог методов чтения свойств (getter), который не нуждается в явном операторе вызова `()`, чтобы получить атрибут экземпляра. На практике это означает, что `a` и `b` будут работать как атрибуты экземпляра, допускающие только чтение, и это имеет смысл, если вспомнить, что экземпляры `DemoNTClass` – просто наделенные дополнительными возможностями кортежи, а кортежи неизменяемы.

`DemoNTClass` получает также строку документации:

```
>>> DemoNTClass.__doc__
'DemoNTClass(a, b)'
```

Исследуем экземпляр `DemoNTClass`:

```
>>> nt = DemoNTClass(8)
>>> nt.a
8 >>> nt.b
1.1
>>> nt.c
'spam'
```

Чтобы сконструировать `nt`, мы должны передать `DemoNTClass` как минимум аргумент `a`. Конструктор принимает также аргумент `b`, но у него есть значение по умолчанию `1.1`, поэтому он необязателен. У объекта `nt` имеются атрибуты `a` и `b`, как и следовало ожидать; атрибута `c` у него нет, но Python берет его из класса, как обычно.

Попытавшись присвоить значения атрибутам `nt.a`, `nt.b`, `nt.c` или даже `nt.z`, мы получим исключения `AttributeError` с несколько различающимися сообщениями об ошибках. Попробуйте сами и поразмыслите об этих сообщениях.

Инспектирование класса с декоратором dataclass

Теперь обратимся к примеру 5.12.

Пример 5.12. meaning/demo_dc.py: класс с декоратором `@dataclass`

```
from dataclasses import dataclass

@dataclass
class DemoDataClass:
    a: int          ❶
    b: float = 1.1  ❷
    c = 'spam'      ❸
```

❶ Для поля `a` заводится аннотация, оно также становится атрибутом экземпляра, управляемым дескриптором.
❷ Для поля `b` также заводится аннотация, и оно также становится атрибутом экземпляра с дескриптором и значением по умолчанию `1.1`.
❸ `c` – самый обычный атрибут класса, никакая аннотация на него не ссылается.

Теперь посмотрим на атрибуты `__annotations__`, `__doc__` и `a`, `b`, `c` класса `DemoDataClass`:

```
>>> from demo_dc import DemoDataClass
>>> DemoDataClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoDataClass.__doc__
'DemoDataClass(a: int, b: float = 1.1)'
>>> DemoDataClass.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'DemoDataClass' has no attribute 'a'
>>> DemoDataClass.b
1.1
>>> DemoDataClass.c
'spam'
```

Атрибуты `__annotations__` и `__doc__` не таят никаких сюрпризов. Но атрибута с именем `a` в классе `DemoDataClass` нет – в отличие от `DemoNTClass` из примера 5.11, в котором имеется дескриптор для получения `a` из экземпляров в виде атрибутов, допускающих только чтение (таинственный атрибут `<_collections._tuplegetter>`). Все объясняется тем, что атрибут `a` существует лишь в экземплярах `DemoDataClass`. Это будет открытый атрибут, который можно читать и записывать, если только класс не заморожен. С другой стороны, `b` и `c` являются атрибутами класса, причем `b` хранит значение по умолчанию для атрибута экземпляра `b`, а `c` – просто атрибут класса, который не будет связан с экземплярами.

Полюбопытствуем, как выглядит экземпляр `DemoDataClass`:

```
>>> dc = DemoDataClass(9)
>>> dc.a
9
>>> dc.b
1.1
>>> dc.c
'spam'
```

И снова `a` и `b` – атрибуты экземпляра, а `c` – атрибут класса, который мы получаем через экземпляр.

Как уже отмечалось, экземпляры `DemoDataClass` изменяемы – и никакой проверки типов на этапе выполнения не производится.

```
>>> dc.a = 10
>>> dc.b = 'oops'
```

Можно даже выполнить еще более глупые присваивания:

```
>>> dc.c = 'whatever'
>>> dc.z = 'secret stash'
```

Теперь у экземпляра `dc` есть атрибут `c`, но присваивание ему не изменяет атрибут класса `c`. И можно добавить новый атрибут `z`. Это обычное поведение Python: у регулярных экземпляров могут быть собственные атрибуты, отсутствующие в классе[1].

Еще о @dataclass

До сих пор мы видели только простые примеры использования `@dataclass`. Этот декоратор принимает несколько именованных аргументов. Вот его сигнатура:

```
@dataclass(*, init=True, repr=True, eq=True, order=False,
             unsafe_hash=False, frozen=False)
```

Символ `*` в первой позиции означает, что остальные параметры чисто именованные. Они описаны в табл. 5.2.

Таблица 5.2. Именованные параметры, принимаемые декоратором `@dataclass`

Параметр	Семантика	Значение по умолчанию	Примечания
`init`	Сгенерировать `__init__`	`True`	Игнорируется, если `__init__` реализован пользователем
`repr`	Сгенерировать `__repr__`	`True`	Игнорируется, если `__repr__` реализован пользователем
`order`	Сгенерировать `__lt__`, `__le__`, `__gt__`, `__ge__`	`False`	Если `True`, то возбуждает исключение, если `eq=False` или любой из подлежащих определению методов сравнения уже определен или унаследован
`unsafe_hash`	Сгенерировать `__hash__`	`False`	Сложная семантика и несколько подвохов; см. документацию по `dataclass` (*https://docs.python.org/3/library/dataclasses.html#dataclasses.dataclass*)
`frozen`	Делать экземпляры «неизменяемыми»	`False`	Экземпляры будут в разумной степени защищены от случайного изменения, но не являются неизменяемыми в полном смысле слова[a]

[a] `@dataclass` эмулирует неизменяемость, генерируя методы `__setattr__` и `__delattr__`, которые возбуждают исключение `dataclass.FrozenInstanceError` – подкласс `AttributeError`, – когда пользователь пытается изменить или удалить поле.

Значения по умолчанию выбраны, исходя из полезности в типичных случаях. Изменять на противоположные, скорее всего, понадобится только значения по умолчанию следующих аргументов:

[1] Задание атрибута после `__init__` сводит на нет оптимизацию памяти `__dict__`, связанную с разделением ключей (см. раздел «Практические последствия внутреннего устройства класса dict» главы 3).

`frozen=True`

Защищает экземпляры класса от непреднамеренного изменения.

`order=True`

Позволяет сортировать экземпляры класса данных.

Учитывая динамическую природу объектов в Python, настырному программисту не составит труда обойти защиту, предоставляемую аргументом `frozen=True`. Но такие трюки легко заметить во время рецензирования кода.

Если оба аргумента, `eq` и `frozen`, равны `True`, то `@dataclass` порождает подходящий метод `__hash__`, так что экземпляры будут допускать хеширование. В сгенерированном методе `__hash__` будут использоваться данные из всех полей, кроме явно исключенных с помощью опции поля, которую мы рассмотрим в разделе «Опции полей» ниже. Если `frozen=False` (по умолчанию), то `@dataclass` установит атрибут `__hash__` равным `None`, сигнализируя о том, что экземпляры не хешируемые, и тем самым отменив метод `__hash__`, унаследованный от суперкласса.

В документе PEP 557 «Data Classes» об аргументе `unsafe_hash` сказано следующее:

> Хотя это и не рекомендуется, вы можете заставить класс данных создать метод `__hash__`, задав аргумент `unsafe_hash=True`. Такое бывает необходимо, если ваш класс логически неизменяемый, но тем не менее может быть изменен. Это очень специальный случай, нуждающийся в тщательном обдумывании.

И это все, что я имею сказать о `unsafe_hash`. Если вам кажется, что без этой возможности не обойтись, почитайте документацию по `dataclasses.dataclass` (*https://docs.python.org/3/library/dataclasses.html#dataclasses.dataclass*).

Дальнейшая настройка сгенерированного класса данных производится на уровне полей.

Опции полей

С самой главной опцией поля мы уже знакомы: предоставлять (или не предоставлять) значение по умолчанию с аннотацией типа. Объявленные нами поля экземпляра становятся параметрами сгенерированного метода `__init__`. Python не позволяет задавать параметр без значения по умолчанию, если до него был задан хотя бы один параметр, имеющий значение по умолчанию. Поэтому если объявлено поле со значением по умолчанию, то и все последующие поля тоже должны иметь значения по умолчанию.

Изменяемые значения по умолчанию – место, где начинающие программисты на Python часто допускают ошибки. В определениях функций изменяемое значение по умолчанию легко запортить, когда при каком-то одном обращении функция изменяет его, после чего изменяется поведение во всех последующих обращениях. Эту проблему мы будем рассматривать в разделе «Значения по умолчанию изменяемого типа: неудачная мысль» главы 6. Атрибуты классов часто используются как значения атрибутов по умолчанию для экземпляров, в т. ч. в классах данных. А декоратор `@dataclass` использует значения по умолчанию в аннотациях типов, чтобы сгенерировать параметры со значениями по умолчанию для `__init__`. Для предотвращения ошибок `@dataclass` отвергает определение класса в примере 5.13.

Пример 5.13. dataclass/club_wrong.py: этот класс возбуждает исключение `ValueError`

```
@dataclass
class ClubMember:
    name: str
    guests: list = []
```

Попытавшись загрузить модуль с таким классом `ClubMember`, мы получим:

```
$ python3 club_wrong.py
Traceback (most recent call last):
  File "club_wrong.py", line 4, in <module>
    class ClubMember:
  ...несколько строк опущено...
ValueError: mutable default <class 'list'> for field guests is not allowed:
use default_factory
```

Сообщение в `ValueError` объясняет, в чем проблема, и предлагает решение: использовать `default_factory`. В примере 5.14 показано, как исправить класс `ClubMember`.

Пример 5.14. dataclass/club.py: это определение `ClubMember` работает

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
    guests: list = field(default_factory=list)
```

В поле `guests` в примере 5.14 мы вместо литерального списка задаем значение по умолчанию путем вызова функции `dataclasses.field` с параметром `default_factory=list`.

Параметр `default_factory` позволяет указать функцию, класс или еще какой-то вызываемый объект, который будет вызываться без аргументов для построения значения по умолчанию при каждом создании экземпляра класса данных. При таком подходе у каждого экземпляра `ClubMember` будет собственный список `list`, а не один `list` на всех, что редко является желаемым поведением и обычно свидетельствует об ошибке.

Хорошо, что `@dataclass` отвергает определения классов со списком в качестве значения поля по умолчанию. Однако следует иметь в виду, что это частичное решение, применимое только к `list`, `dict` и `set`. Про другие изменяемые значения, использованные в качестве значений по умолчанию, `@dataclass` ничего не говорит. Вы сами должны понимать суть проблемы и пользоваться фабрикой, когда нужно задать изменяемое значение по умолчанию.

Покопавшись в документации по модулю `dataclasses` (*https://docs.python.org/3/library/dataclasses.html*), вы наткнетесь на определение поля типа `list` с новым синтаксисом, показанным в примере 5.15.

Пример 5.15. dataclass/club_generic.py: это определение `ClubMember` более точное

```
from dataclasses import dataclass, field

@dataclass
```

```
class ClubMember:
    name: str
    guests: list[str] = field(default_factory=list) ❶
```

❶ `list[str]` означает «список объектов типа `str`».

Новый синтаксис `list[str]` – это параметризованный обобщенный тип: начиная с версии Python 3.9 встроенный тип `list` допускает задание типа элементов списка в квадратных скобках.

До Python 3.9 встроенные коллекции не поддерживали нотацию обобщенного типа. В качестве временного обходного решения в модуле `typing` имеются соответствующие типы коллекций. Если в версии Python 3.8 или более ранней вам нужен был тип параметризованного списка, то следовало импортировать тип `List` из `typing` и пользоваться им: `List[str]`. Дополнительные сведения на эту тему см. во врезке «Поддержка унаследованных типов и нерекомендуемые типы коллекций» в главе 8.

Обобщенные типы мы будем рассматривать в главе 8. А пока заметим, что оба примера, 5.14 и 5.15, правильны и программа проверки типов Муру не ругается ни на одно из этих определений класса.

Разница в том, что `guests: list` означает, что `guests` может быть списком объектов любого типа, а `guests: list[str]` говорит, что `guests` должно быть списком строк. Это позволяет программе проверки типов находить некоторые ошибки в коде, пытающемся поместить недопустимые элементы в список или прочитать то, чего в списке не может быть.

`default_factory` – пожалуй, самая часто используемая опция функции `field`, но есть и другие. Все они перечислены в табл. 5.3.

Таблица 5.3. Именованные аргументы, принимаемые функцией `field`

Опция	Семантика	Значение по умолчанию
`default`	Значение поля по умолчанию	`_MISSING_TYPE`[a]
`default_factory`	Функция без параметров, порождающая значение по умолчанию	`_MISSING_TYPE`
`init`	Включить поле в состав параметров `__init__`	`True`
`repr`	Включить поле в `__repr__`	`True`
`compare`	Использовать поле в методах сравнения `__eq__`, `__lt__` и т. д.	`True`
`hash`	Включить поле в вычисление `__hash__`	`None`[b]
`metadata`	Отображение, содержащее пользовательские данные; игнорируется `@dataclass`	`None`

[a] `dataclass._MISSING_TYPE` – это специальное значение, означающее, что опция не задана. Существует для того, чтобы можно было задать `None` в качестве фактического значения по умолчанию, как часто бывает.

[b] Опция `hash=None` означает, что поле будет использоваться при вычислении `__hash__`, только если `compare=True`.

Опция `default` существует, потому что обращение к `field` заменяет значение по умолчанию в аннотации поля. Если мы хотим создать поле `athlete` со значением по умолчанию `False`, но не включать это поле в метод `__repr__`, то должны будем написать такое определение:

```
@dataclass
class ClubMember:
    name: str
    guests: list = field(default_factory=list)
    athlete: bool = field(default=False, repr=False)
```

Постинициализация

Метод `__init__`, генерируемый декоратором `@dataclass`, принимает только переданные аргументы и присваивает их (или их значения по умолчанию в случае отсутствия) атрибутам экземпляра, которые являются полями экземпляра. Но иногда инициализировать экземпляр недостаточно. В таком случае можно предоставить метод `__post_init__`. Если он существует, то `@dataclass` добавляет в сгенерированный метод `__init__` обращение к `__post_init__` в качестве последнего шага.

Типичные сценарии использования `__post_init__` – проверка и вычисление значений полей на основе других полей. Мы изучим простые примеры применения `__post_init__` по обеим причинам.

Сначала рассмотрим ожидаемое поведение подкласса `HackerClubMember` класса `ClubMember`, поведение которого описывается тестами в примере 5.16.

Пример 5.16. dataclass/hackerclub.py: тесты для `HackerClubMember`

```
"""
``HackerClubMember`` объект принимает необязательный аргумент ``handle``::

    >>> anna = HackerClubMember('Anna Ravenscroft', handle='AnnaRaven')
    >>> anna
    HackerClubMember(name='Anna Ravenscroft', guests=[], handle='AnnaRaven')

Если ``handle`` опущен, то берется первая часть имени члена::

    >>> leo = HackerClubMember('Leo Rochael')
    >>> leo
    HackerClubMember(name='Leo Rochael', guests=[], handle='Leo')

Члены должны иметь уникальный описатель. Следующий объект ``leo2`` не будет
создан потому, что его описатель ``handle`` был бы равен 'Leo', но этот описатель
уже сопоставлен ``leo``::

    >>> leo2 = HackerClubMember('Leo DaVinci')
    Traceback (most recent call last):
      ...
    ValueError: handle 'Leo' already exists.

Чтобы исправить эту ошибку, ``leo2`` необходимо создать с явным описателем::

    >>> leo2 = HackerClubMember('Leo Dainci', handle='Neo')
    >>> leo2
    HackerClubMember(name='Leo DaVinci', guests=[], handle='Neo')
"""
```

Заметим, что задавать `handle` следует в виде именованного аргумента, потому что `HackerClubMember` наследует `name` и `guests` от `ClubMember` и добавляет поле `handle`. Сгенерированная строка документации для `HackerClubMember` показывает порядок полей при вызове конструктора:

```
>>> HackerClubMember.__doc__
"HackerClubMember(name: str, guests: list = <factory>, handle: str = '')"
```

Здесь `<factory>` – короткий способ сказать, что некоторый вызываемый объект порождает значение по умолчанию для `guests` (в нашем случае фабрика – это класс `list`). Мораль: чтобы задать `handle`, но не задавать `guests`, мы должны передать `handle` в качестве именованного аргумента.

В разделе «Наследование» документации по модулю `dataclasses` (*https://docs.python.org/3/library/dataclasses.html#inheritance*) объясняется, как вычисляется порядок полей при наличии нескольких уровней наследования.

В главе 14 мы будем говорить о неправильном использовании наследования, особенно когда суперклассы не являются абстрактными. Создавать иерархию классов обычно не рекомендуется, но тут это сослужило нам добрую службу, позволив сократить пример 5.17 и сосредоточиться на объявлении поля `handle` и проверке в методе `__post_init__`.

В примере 5.17 показана реализация.

Пример 5.17. dataclass/hackerclub.py: код класса `HackerClubMember`

```
from dataclasses import dataclass
from club import ClubMember

@dataclass
class HackerClubMember(ClubMember): ❶
    all_handles = set() ❷
    handle: str = '' ❸

    def __post_init__(self):
        cls = self.__class__ ❹
        if self.handle == '': ❺
            self.handle = self.name.split()[0]
        if self.handle in cls.all_handles: ❻
            msg = f'handle {self.handle!r} already exists.'
            raise ValueError(msg)
        cls.all_handles.add(self.handle) ❼
```

❶ `HackerClubMember` расширяет `ClubMember`.
❷ `all_handles` – атрибут класса.
❸ `handle` – поле экземпляра, имеющее тип `str` и по умолчанию равное пустой строке; это делает его факультативным.
❹ Получить класс экземпляра.
❺ Если `self.handle` – пустая строка, положить ее равной первой части `name`.
❻ Если `self.handle` уже присутствует в `cls.all_handles`, возбудить исключение `ValueError`.
❼ Добавить новый `handle` в `cls.all_handles`.

Пример 5.17 работает, как и ожидается, но программа статической проверки типов с ним не справится. Далее мы объясним, почему и как это исправить.

Типизированные атрибуты класса

Если проверить пример 5.17 с помощью Муру, то мы получим сообщение об ошибке:

```
$ mypy hackerclub.py
hackerclub.py:37: error: Need type annotation for «all_handles»
(hint: «all_handles: Set[<type>] = ...»)
Found 1 error in 1 file (checked 1 source file)
```

К сожалению, решение, предлагаемое Муру (когда я писал этот раздел, текущей версией была 0.910), бесполезно в контексте применения `@dataclass`. Во-первых, предлагается использовать `Set`, но я работаю с Python 3.9, поэтому могу применить `set` – и избежать импорта класса `Set` из модуля `typing`. Но важнее другое – если добавить аннотацию вида `set[…]` к `all_handles`, то `@dataclass` увидит ее и сделает `all_handles` атрибутом экземпляра. К чему это приводит, мы видели в разделе «Инспектирование класса с декоратором dataclass» выше.

Обходной путь, описанный в документе PEP 526 «Syntax for Variable Annotations» (*https://peps.python.org/pep-0526/#class-and-instance-variable-annotations*), никак не назовешь элегантным. Чтобы завести переменную класса с аннотацией типа, мы должны использовать псевдотип `typing.ClassVar`, в котором применяется нотация обобщенных типов `[]`, дабы задать тип переменной и одновременно объявить ее атрибутом класса.

Чтобы удовлетворить программу проверки типов и декоратор `@dataclass`, предлагается объявить `all_handles` в примере 5.17 следующим образом:

```
all_handles: ClassVar[set[str]] = set()
```

Вот что означает эта аннотация типа:

> *`all_handles` – атрибут типа `set-of-str`, для которого значением по умолчанию является пустое множество.*

Чтобы закодировать эту аннотацию, мы должны импортировать `ClassVar` из модуля `typing`.

Декоратору `@dataclass` безразличны типы в аннотациях, за исключением двух случаев. Один из них такой: если тип атрибута равен `ClassVar`, то для него не генерируется поле экземпляра.

Второй случай, когда тип поля имеет значение для `@dataclass`, – объявление *переменных только для инициализации*. Мы рассмотрим его в следующем разделе.

Инициализируемые переменные, не являющиеся полями

Иногда возникает необходимость передать `__init__` аргументы, не являющиеся полями экземпляра. В документации по `dataclasses` (*https://docs.python.org/3/library/dataclasses.html#init-only-variables*) они называются *переменными только для инициализации*. Для объявления такого аргумента модуль `dataclasses` предоставляет псевдотип `InitVar`, в котором используется тот же синтаксис, что и в `typing.ClassVar`. В документации приведен пример – класс данных, в котором

имеется поле, инициализируемое из базы данных, поэтому конструктору необходимо передать объект базы данных.

В примере 5.18 показан код, иллюстрирующий использование переменных только для инициализации.

Пример 5.18. Пример из документации по модулю `dataclasses`

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

Обратите внимание, как объявлен атрибут `database`. `InitVar` не дает декоратору `@dataclass` обращаться с `database` как с обычным полем. Он не будет создавать для него атрибут экземпляра, а функция `dataclasses.fields` не включит его в список полей. Однако `database` станет одним из аргументов, принимаемых сгенерированным `__init__`, и будет передаваться методу `__post_init__`. Если вы будете писать этот метод, то должны добавить соответствующий аргумент в его сигнатуру, как показано в примере 5.18.

В этом довольно пространном обзоре декоратора `@dataclass` мы остановились на наиболее полезных средствах. Некоторые из них уже были упомянуты в предыдущих разделах, например «Основные возможности», где параллельно рассматривались все три построителя классов данных. Полное изложение смотрите в документации по `dataclasses` и в документе PEP 526 «Syntax for Variable Annotations».

В следующем разделе я представлю более длинный пример использования `@dataclass`.

Пример использования @dataclass: запись о ресурсе из дублинского ядра

Часто классы, построенные с помощью `@dataclass`, включают больше полей, чем во встречавшихся до сих пор коротеньких примерах. Дублинское ядро (*https://www.dublincore.org/specifications/dublin-core/*) дает материал для более представительного примера использования `@dataclass`.

> Схема дублинского ядра – это небольшой набор словарных терминов, которые можно использовать для описания цифровых ресурсов (видео, изображений, веб-страниц и т. д.), а также физических ресурсов: книг, компакт-дисков и предметов искусства[1].
>
> – Дублинское ядро в Википедии

В стандарте определено 15 факультативных полей, в классе `Resource` из примера 5.19 используются восемь из них.

[1] Источник: статья о дублинском ядре в Википедии (*https://en.wikipedia.org/wiki/Dublin_Core*).

Пример 5.19. dataclass/resource.py: код класса `Resource`, основанного на схеме дублинского ядра

```
from dataclasses import dataclass, field
from typing import Optional
from enum import Enum, auto
from datetime import date

class ResourceType(Enum): ❶
    BOOK = auto()
    EBOOK = auto()
    VIDEO = auto()

@dataclass
class Resource:
    """ Описание мультимедийного ресурса."""
    identifier: str ❷
    title: str = '<untitled>' ❸
    creators: list[str] = field(default_factory=list)
    date: Optional[date] = None ❹
    type: ResourceType = ResourceType.BOOK ❺
    description: str = ''
    language: str = ''
    subjects: list[str] = field(default_factory=list)
```

❶ Это перечисление `Enum` содержит допустимые значения поля `Resource.type`.
❷ `identifier` – единственное обязательное поле.
❸ `title` – первое поле со значением по умолчанию. Это значит, что все последующие поля тоже должны иметь значения по умолчанию.
❹ Значение `date` может быть экземпляром `datetime.date` или `None`.
❺ Для поля `type` значением по умолчанию является `ResourceType.BOOK`.

В примере 5.20 приведен тест, показывающий, как запись типа `Resource` должна использоваться в коде.

Пример 5.20. Тест класса `Resource`

```
>>> description = 'Improving the design of existing code'
>>> book = Resource('978-0-13-475759-9', 'Refactoring, 2nd Edition',
...     ['Martin Fowler', 'Kent Beck'], date(2018, 11, 19),
...     ResourceType.BOOK, description, 'EN',
...     ['computer programming', 'OOP'])
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(identifier='978-0-13-475759-9', title='Refactoring, 2nd Edition',
creators=['Martin Fowler', 'Kent Beck'],
date=datetime.date(2018, 11, 19),
type=<ResourceType.BOOK: 1>, description='Improving the design of existing code',
language='EN', subjects=['computer programming', 'OOP'])
```

Метод `__repr__`, сгенерированный `@dataclass`, годится, но можно сделать представление более удобочитаемым. Вот какого формата мы ждем от `repr(book)`:

```
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(
    identifier = '978-0-13-475759-9',
```

```
        title = 'Refactoring, 2nd Edition',
        creators = ['Martin Fowler', 'Kent Beck'],
        date = datetime.date(2018, 11, 19),
        type = <ResourceType.BOOK: 1>,
        description = 'Improving the design of existing code',
        language = 'EN',
        subjects = ['computer programming', 'OOP'],
    )
```

В примере 5.21 приведен код `__repr__`, порождающий такой формат. Здесь используется функция `dataclass.fields`, которая возвращает имена всех полей класса данных.

Пример 5.21. dataclass/resource_repr.py: код метода `__repr__`, реализованного в классе `Resource` из примера 5.19

```
    def __repr__(self):
        cls = self.__class__
        cls_name = cls.__name__
        indent = ' ' * 4
        res = [f'{cls_name}('] ❶
        for f in fields(cls): ❷
            value = getattr(self, f.name) ❸
            res.append(f'{indent}{f.name} = {value!r},') ❹
        res.append(')') ❺
        return '\n'.join(res) ❻
```

❶ Начать список `res` для построения выходной строки. Включить имя класса и открывающую скобку.
❷ Для каждого поля `f` класса ...
❸ ... получить именованный атрибут из экземпляра.
❹ Дописать в конец начинающуюся отступом строку, содержащую имя поля и `repr(value)`, – именно это делает `!r`.
❺ Дописать закрывающую скобку.
❻ Построить по `res` многострочную строку и вернуть ее.

Этим примером мы завершаем знакомство с построителями классов данных в Python.

Классы данных удобны, но если использовать их неумеренно, проект может пострадать. В следующем разделе объясняется, в чем проблема.

Класс данных как признак кода с душком

Не важно, пишете вы код класса данных самостоятельно или пользуетесь одним из описанных выше построителей классов, необходимо понимать, что само его наличие может быть индикатором проблемы в дизайне.

Во втором издании книги «Рефакторинг»[1] Мартин Фаулер и Кент Бек представили каталог примеров «дурно пахнущего кода» – паттернов, которые указывают на необходимость рефакторинга. Раздел, озаглавленный «Класс данных», начинается так:

[1] *М. Фаулер, К. Бек.* Рефакторинг. Диалектика-Вильямс, 2019.

> Это классы, в которых имеются поля, методы чтения и записи этих полей и больше ничего. Такие классы – тупые хранители данных, и зачастую другие классы манипулируют ими, вникая в слишком большое число деталей.

На персональном сайте Фаулера имеется статья «Code Smell» (*https://martinfowler.com/bliki/CodeSmell.html*), проясняющая эту точку зрения. Она имеет непосредственное отношение к нашему обсуждению, потому что *класс данных* используется как один из примеров кода с душком и даются рекомендации, что с этим делать. Приведу полный текст статьи[1].

Код с душком

Мартин Фаулер

Код с душком (code smell) – это видимый признак, который обычно соответствует более глубокой проблеме в системе. Термин предложил Кент Бек, помогавший мне при написании книги «Рефакторинг» (*https://martinfowler.com/books/refactoring.html*).

Данное выше краткое определение содержит два тонких момента. Во-первых, душок, по определению, – нечто такое, что легко обнаружить, или унюхать, как я с недавних пор стал это называть. Длинный метод – прекрасный пример; стоит мне увидеть, что в методе на Java больше дюжины строк, как нос тут же начинается чесаться.

Второй момент – душок необязательно является верным признаком проблемы. Иногда длинный метод – это нормально. Нужно заглянуть глубже и понять, действительно ли проблема существует, – запахи сами по себе не плохи, они являются индикатором проблемы, а не самой проблемой.

Самые лучшие запахи – те, что легко обнаружить, и, как правило, они свидетельствуют о по-настоящему интересных проблемах. Классы данных (содержащие только данные и никакого поведения) могут служить отличным примером. Мы смотрим на них и задаемся вопросом, какого поведения здесь не хватает. А потом приступаем к рефакторингу, чтобы перенести нужное поведение в класс. Часто простые вопросы и начальный рефакторинг могут стать важным шагом на пути превращения анемичных объектов в нечто, заслуживающее называться классом.

У кода с душком есть полезная черта – его может выявить даже неопытный человек, не имеющий достаточно знаний, чтобы понять, существует ли реальная проблема и как ее исправить. Я слышал о руководителях группы, которые выбирали «душок недели» и просили коллег поискать код с таким душком и представить его на рассмотрение старших членов группы. Разбираться с одним душком за раз – отличный способ постепенно обучать членов группы навыкам хорошего программирования.

Основная идея объектно-ориентированного программирования – собрать данные и поведение в одной единице кода: классе. Если класс широко используется, но не имеет собственного поведения, то может статься, что код, работающий с его экземплярами, разбросан (и даже дублируется) по разным мето-

[1] Мне повезло работать вместе с Мартином Фаулером в компании Thoughtworks, так что на получение разрешения ушло всего 20 минут.

дам и функциям. А это неизбежно ведет к трудностям на этапе сопровождения. Поэтому рефакторинг, предлагаемый Фаулером в отношении классов данных, призван вернуть ему положенные обязанности.

С учетом сказанного все же имеется два типичных сценария, когда имеет смысл завести класс данных, обделенный поведением.

Класс данных как временная конструкция

В этом случае класс данных является начальной упрощенной реализацией некоторого класса, необходимой, чтобы приступить к работе над новым проектом или модулем. Со временем класс получит собственные методы и перестанет полагаться на методы других классов для работы со своими экземплярами. Это можно сравнить со строительными лесами – они временные, а в конечном итоге класс, возможно, перестанет зависеть от построителя, с которого все начиналось.

Кроме того, Python нередко используется для быстрого решения задачи и экспериментов, и тогда можно оставить временную конструкцию навсегда.

Класс данных как промежуточное представление

Класс данных может быть полезен и для построения записей, которые впоследствии экспортируются в формате JSON или еще каком-то формате обмена данными, либо для хранения только что импортированных данных, пересекающих какие-то границы внутри системы. Все построители классов данных в Python предоставляют метод или функцию для преобразования экземпляра в простой словарь `dict`. Кроме того, всегда можно вызвать конструктор, передав ему словарь именованных аргументов, расширяемый с помощью оператора `**`. Такой словарь очень близок к записи в формате JSON.

В этом случае к экземплярам класса данных следует относиться как к неизменяемым объектам; даже если поля изменяемые, все равно не следует изменять их, пока они пребывают в такой промежуточной форме. Если вы не последуете этому совету, то потеряете главное преимущество от размещения данных и поведения в одном месте. Если при импорте или экспорте необходимо изменять значения, то следует реализовать собственные методы построителя, не используя ни предлагаемые методы типа «as dict», ни стандартные конструкторы.

Теперь сменим тему и посмотрим, как писать образцы, с которыми можно сопоставлять экземпляры произвольных классов, а не только последовательности и отображения, о которых мы говорили в разделах «Сопоставление с последовательностями-образцами» и «Сопоставление с отображениями-образцами» главы 3.

Сопоставление с экземплярами классов – образцами

Классы-образцы предназначены для сопоставления с экземплярами классов по типу и факультативно по атрибутам. Субъектом класса-образца может быть экземпляр любого класса, а не только класса данных[1].

[1] Я поместил этот материал сюда, потому что это первая из глав, где речь идет о пользовательских классах, и подумал, что тема сопоставления с классами-образцами настолько важна, что не стоит откладывать ее до второй части книги. Я руководствуюсь следующим принципом: важнее знать, как использовать классы, чем как их определять.

Существует три вида классов-образцов: простой, именованный и позиционный. Мы рассмотрим их в этом порядке.

Простые классы-образцы

В разделе «Сопоставление с последовательностями-образцами» мы уже видели, как простые классы-образцы используются в качестве подобразцов:

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

С этим образцом сопоставляется четырехэлементная последовательность, в которой первый элемент должен быть экземпляром класса `str`, а последний – 2-кортежем, содержащим два экземпляра `float`.

Синтаксически классы-образцы выглядят как вызов конструктора. Ниже показан класс-образец, с которым сопоставляются значения `float` без связывания с переменной (в теле `case` можно при необходимости ссылаться на `x` напрямую):

```
match x:
    case float():
        do_something_with(x)
```

Но следующий пример, скорее всего, содержит ошибку:

```
match x:
    case float:  # ОПАСНО!!!
        do_something_with(x)
```

В этом примере `case float:` сопоставляется с любым субъектом, потому что Python рассматривает `float` как переменную, которая затем связывается с субъектом.

Синтаксис простого образца `float(x)` – специальный случай, применимый только к девяти получившим особое благословение встроенным типам, перечисленным в конце раздела «Class Patterns» (*https://peps.python.org/pep-0634/#class-patterns*) документа PEP 634 «Structural Pattern Matching: Specification»:

```
bytes dict float frozenset int list set str tuple
```

В этих классах переменная, которая выглядит как аргумент конструктора, например `x` в `float(x)`, связывается со всем экземпляром субъекта или его частью, которая сопоставляется с подобразцом, как продемонстрировано в рассмотренном ранее примере сопоставления с последовательностью-образцом:

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

Если класс не является одним из этих привилегированных встроенных классов, то похожие на аргумент переменные представляют образцы, с которыми должны сопоставляться атрибуты экземпляра класса.

Именованные классы-образцы

Чтобы понять, как используются именованные классы-образцы, рассмотрим следующий класс `City` и пять его экземпляров в примере 5.22.

Пример 5.22. Класс `City` и несколько его экземпляров

```
import typing

class City(typing.NamedTuple):
    continent: str
    name: str
    country: str

cities = [
    City('Asia', 'Tokyo', 'JP'),
    City('Asia', 'Delhi', 'IN'),
    City('North America', 'Mexico City', 'MX'),
    City('North America', 'New York', 'US'),
    City('South America', 'São Paulo', 'BR'),
]
```

При таких определениях следующая функция вернет список азиатских городов:

```
    def match_asian_cities():
        results = []
        for city in cities:
            match city:
                case City(continent='Asia'):
                    results.append(city)
        return results
```

С образцом `City(continent='Asia')` сопоставляется любой экземпляр `City`, в котором значение атрибута `continent` равно `'Asia'`, вне зависимости от значений других атрибутов.

Чтобы собрать в коллекцию значения атрибута `country`, можно было бы написать:

```
    def match_asian_countries():
        results = []
        for city in cities:
            match city:
                case City(continent='Asia', country=cc):
                    results.append(cc)
            return results
```

С образцом `City(continent='Asia', country=cc)` сопоставляются те же азиатские города, что и раньше, но теперь переменная `cc` связана с атрибутом `country` каждого экземпляра. Это работает и тогда, когда переменная-образец тоже называется `country`:

```
            match city:
                case City(continent='Asia', country=country):
                    results.append(country)
```

Именованные классы-образцы прекрасно воспринимаются на глаз и работают с любым классом, имеющим открытые атрибуты экземпляра. Но они несколько многословны.

Позиционные классы-образцы в некоторых случаях более удобны, но требуют явной поддержки со стороны класса субъекта, как будет показано ниже.

Позиционные классы-образцы

При тех же определениях, что в примере 5.22, следующая функция вернет список азиатских городов, только сопоставление производится с позиционным классом-образцом:

```
def match_asian_cities_pos():
    results = []
    for city in cities:
        match city:
            case City('Asia'):
                results.append(city)
        return results
```

С образцом `City('Asia')` сопоставляется любой экземпляр `City`, в котором первый атрибут равен `'Asia'`, вне зависимости от значений других атрибутов.

Чтобы собрать в коллекцию значения атрибута `country`, можно было бы написать:

```
def match_asian_countries_pos():
    results = []
    for city in cities:
        match city:
            case City('Asia', _, country):
                results.append(country)
    return results
```

С образцом `City('Asia', _, country)` сопоставляются те же города, что и раньше, но теперь переменная `country` связана с третьим атрибутом экземпляра.

Я сказал «первый атрибут», «третий атрибут», но что это в действительности означает?

Класс `City` или любой другой класс может работать с позиционными образцами, только если в нем присутствует специальный атрибут класса с именем `__match_args__`, который автоматически создают построители классов, рассматриваемые в этой главе. Вот как выглядит атрибут `__match_args__` в классе `City`:

```
>>> City.__match_args__
('continent', 'name', 'country')
```

Как видите, `__match_args__` объявляет имена атрибутов в том порядке, в котором они будут использоваться в позиционных образцах.

В разделе «Поддержка сопоставления с позиционными образцами» главы 11 мы напишем код, в котором атрибут класса `__match_args__` создается без участия построителя классов.

Можно сочетать именованные и позиционные аргументы в одном образце. Можно включать в `__match_args__` не все атрибуты экземпляра. Поэтому иногда мы просто вынуждены использовать в образце именованные аргументы в дополнение к позиционным.

Пришло время подводить итоги.

Резюме

Основной темой этой главы были построители классов данных `collections.namedtuple`, `typing.NamedTuple` и `dataclasses.dataclass`. Мы видели, что каждый из них генерирует классы данных по описаниям, предоставленным в виде аргументов фабричной функции, или по предложениям `class` с аннотациями типов в последних двух случаях. В частности, оба варианта на основе именованных кортежей порождают подклассы `tuple`, добавляя лишь возможность обращаться к полям по имени и предоставление атрибута класса `_fields`, в котором имена полей перечисляются в виде кортежа строк.

Затем мы сравнили основные возможности всех трех построителей классов, в частности как извлекать данные экземпляра в виде словаря, как получать имена полей и их значения по умолчанию и как создавать новый экземпляр из существующего.

Это стало предлогом для первого знакомства с аннотациями типов, особенно с теми, что применяются для аннотирования атрибутов в предложении `class` с использованием нотации, предложенной в документе PEP 526 «Syntax for Variable Annotations» (*https://peps.python.org/pep-0526/*) и впервые реализованной в версии Python 3.6. Быть может, самый удивительный аспект аннотаций типов вообще – тот факт, что они не оказывают никакого влияния на этапе выполнения. Python остается динамическим языком. Чтобы воспользоваться преимуществами информации о типах для обнаружения ошибок путем статического анализа исходного кода, необходимы внешние инструменты, например Муру. После краткого обзора синтаксиса, описанного в PEP 526, мы перешли к изучению аннотаций в обычном классе и в классах, построенных с помощью `typing.NamedTuple` и `@dataclass`.

Затем рассмотрели наиболее востребованные средства декоратора `@dataclass` и опцию `default_factory` функции `dataclasses.field`. Мы также уделили внимание аннотациям специальных псевдотипов `typing.ClassVar` и `dataclasses.InitVar`, которые важны в контексте классов данных. Изложение основного предмета главы завершилось примером на основе схемы дублинского ядра, что позволило проиллюстрировать использование `dataclasses.fields` для перебора атрибутов экземпляра `Resource` в пользовательском методе `__repr__`.

Далее мы предостерегли от возможного злоупотребления классами данных, нарушающего принцип объектно-ориентированного программирования: данные и работающие с ними функции должны находиться в одном и том же классе. Классы, не содержащие никакой логики, могут быть признаком того, что логика помещена не туда, где должна быть.

В последнем разделе мы видели, как сопоставление с образцом работает, когда субъект является экземпляром произвольного класса, а не только класса, порожденного рассмотренными в этой главе построителями.

Дополнительная литература

Стандартная документация Python по построителям классов данных написана очень хорошо и содержит немало небольших примеров.

В частности, что касается декоратора `@dataclass`, в документацию по модулю `dataclasses` (*https://docs.python.org/3/library/dataclasses.html*) скопирована бóльшая часть документа PEP 557 «Data Classes» (*https://peps.python.org/pep-0557/*). Но в этом документе имеется несколько весьма информативных разделов, которые не были скопированы, в т. ч. «Why not just use namedtuple?» (Почему бы просто не использовать именованный кортеж), «Why not just use typing.NamedTuple?» (Почему бы просто не использовать typing.NamedTuple) и «Rationale» (Обоснование). Последний заканчивается таким вопросом и ответом на него:

> Когда не стоит использовать классы данных?
> Требуется совместимость API с кортежами или словарями. Требуется проверка типов, выходящая за рамки документов PEP 484 и 526, либо проверка или преобразование значений.
>
> – Eric V. Smith, PEP 557 «Rationale»

На сайте *RealPython.com* Гейр Арне Ньелле (Geir Arne Hjelle) написал очень полное «Пособие по классам данных в Python 3.7» (*https://realpython.com/python-data-classes/*).

На конференции PyCon US 2018 Раймонд Хэттингер провел презентацию «Dataclasses: The code generator to end all code generators» (*https://www.youtube.com/watch?v=T-TwcmT6Rcw*).

Если говорить о дополнительных возможностях и продвинутой функциональности, включая проверку, то несколькими годами ранее `dataclasses` появился проект *attrs* (*https://www.attrs.org/en/stable/*) под руководством Хинека Шлавака, который обещает «вернуть радость написания классов, освободив автора от рутины реализации объектных протоколов (они же dunder-методы)». Эрик В. Смит, автор документа PEP 557, признает влияние проекта *attrs* на дизайн декоратора `@dataclass`. Возможно, это относится и к самому важному решению Смита относительно API: использовать для решения задачи декоратор класса вместо базового класса и (или) метакласса.

Глиф (Glyph), основатель проекта Twisted, написал великолепное введение в *attrs* в статье «The One Python Library Everyone Needs» (*https://glyph.twistedmatrix.com/2016/08/attrs.html*). Документация по проекту *attrs* содержит обсуждение альтернатив (*https://www.attrs.org/en/stable/why.html*).

Автор книг, преподаватель и одержимый ученый-компьютерщик Дэйв Бизли написал *cluegen* (*https://github.com/dabeaz/cluegen*), еще один генератор классов данных. Если вы слушали какие-нибудь презентации Дэйва, то знаете, что он большой и убежденный мастер метапрограммирования на Python. Поэтому для меня источником вдохновения стали изложенные в файле *README.md* мысли о конкретном сценарии, который побудил его написать альтернативу имеющемуся в Python декоратору `@dataclass`, и его философия предоставления подхода к решению задачи в противоположность инструменту: поначалу может показаться, что использовать инструмент быстрее, но подход обладает большей гибкостью и может доставить вас настолько далеко, насколько вы расположены идти.

Что касается отношения к *классу данных* как к коду с душком, то лучший из известных мне источников – второе издание книги Мартина Фаулера «Рефакторинг». В этом последнем издании отсутствует цитата, вынесенная в эпиграф к этой главе: «Классы данных – как дети…», но в остальном это лучшее издание

самой знаменитой книги Фаулера, особенно для питонистов, поскольку примеры написаны на современном JavaScript, который ближе к Python, чем Java – язык, использованный в первом издании.

На сайте *Refactoring Guru* также имеется описание душка, распространяемого кодом классов данных (*https://refactoring.guru/smells/data-class*).

Поговорим

В глоссарии на сайте The Jargon File статья «Guido» (*https://web.archive.org/web/20190204130328/http://catb.org/esr/jargon/html/G/Guido.html*) посвящена Гвидо ван Россуму. Среди прочего в ней есть такие слова:

> Есть поверье, что самым важным атрибутом Гвидо, если не считать самого Python, является его машина времени, которой, говорят, он обладает, потому что снова и снова на запросы пользователей о новых возможностях следует ответ «А я как раз вчера вечерком это реализовал...».

Очень долго в синтаксисе Python не было одной важной части – быстрого и стандартного способа создавать атрибуты экземпляров в классе. Во многих объектно-ориентированных языках такая возможность была. Вот часть определения класса `Point` в Smalltalk:

```
Object subclass: #Point
    instanceVariableNames: 'x y'
    classVariableNames: ''
    package: 'Kernel-BasicObjects'
```

Во второй строке перечисляются имена атрибутов экземпляра `x` и `y`. Если бы это были атрибуты класса, то они находились бы в третьей строке.

Python всегда предлагал простой способ объявления атрибутов класса, имеющих начальное значение. Но атрибуты экземпляра встречаются куда чаще, и программисты были вынуждены заглядывать в метод `__init__`, чтобы найти их, постоянно опасаясь, нет ли еще каких-то атрибутов, определенных в другом месте класса или даже внешними функциями или методами других классов.

А теперь у нас есть `@dataclass`, ура!

Но вместе с ним пришли новые проблемы.

Во-первых, если используется `@dataclass`, то аннотации типов перестают быть факультативными. Последние семь лет, с момента выхода документа PEP 484 «Type Hints» (*https://peps.python.org/pep-0484/*), нам обещали, что аннотации всегда будут необязательными. А теперь мы имеем важное новое средство языка, которое без них работать не может. Если вам не нравится весь этот уклон в сторону статической типизации, то можете вместо этого пользоваться *attrs* (*https://www.attrs.org/en/stable/*).

Во-вторых, синтаксис, предлагаемый в PEP 526 для аннотирования атрибутов класса и экземпляра, выворачивает наизнанку установившееся соглашение о предложениях `class`: все, что объявлено на верхнем уровне блока `class`, является атрибутом класса (методы – тоже атрибуты класса). А в PEP 526 и `@dataclass` всякий атрибут, объявленный на верхнем уровне с аннотацией типа, становится атрибутом экземпляра:

```
@dataclass
class Spam:
    repeat: int # атрибут экземпляра
```

Во фрагменте ниже `repeat` – тоже атрибут экземпляра:

```
@dataclass
class Spam:
    repeat: int = 99 # атрибут экземпляра
```

Но если аннотаций типов нет, то мы внезапно возвращаемся в старые добрые времена, когда объявления, находящиеся на верхнем уровне класса, принадлежали самому классу, а не его экземплярам:

```
@dataclass
class Spam:
    repeat = 99 # атрибут класса!
```

Наконец, если мы хотим аннотировать атрибут класса типом, то не можем использовать регулярные типы, потому что тогда получим атрибут экземпляра. И приходится прибегать к аннотации псевдотипа `ClassVar`:

```
@dataclass
class Spam:
    repeat: ClassVar[int] = 99 # ppppp!
```

Здесь мы имеем исключение из исключения из правила. По мне – так в высшей степени антипитонично.

Я не принимал участия в обсуждениях, закончившихся появлением документов PEP 526 и PEP 557 «Data Classes» (*https://peps.python.org/pep-0557/*), но хотел бы видеть другой синтаксис, а именно:

```
@dataclass
class HackerClubMember:
    .name: str ❶
    .guests: list = field(default_factory=list)
    .handle: str = ''
    all_handles = set() ❷
```

❶ Атрибуты экземпляра должны быть объявлены с префиксом `.`.

❷ Любое имя атрибута без префикса `.` считается атрибутом класса (как всегда и было).

Для реализации этого предложения пришлось бы изменить грамматику языка. Мне такой синтаксис кажется вполне понятным, и не возникает никаких исключений из исключений.

Хотелось бы мне одолжить машину времени Гвидо, чтобы вернуться в 2017 год и продать эту идею команде разработчиков ядра.

Глава 6

Ссылки на объекты, изменяемость и повторное использование

> – Ты загрустила? – огорчился Рыцарь. – Давай я спою тебе в утешение песню.
> [...] Заглавие этой песни называется «ПУГОВКИ ДЛЯ СЮРТУКОВ».
> – Вы хотите сказать – песня так называется? – спросила Алиса, стараясь заинтересоваться песней.
> – Нет, ты не понимаешь, – ответил нетерпеливо Рыцарь. – Это ЗАГЛАВИЕ так называется. А песня называется «ДРЕВНИЙ СТАРИЧОК».
>
> – Льюис Кэрролл, «Алиса в Зазеркалье»

Алиса и Рыцарь задают тон тому, о чем пойдет речь в этой главе. Ее тема – различие между объектами и их именами. Имя – это не объект, а совершенно отдельная вещь.

Мы начнем главу с метафоры переменных в Python: переменные – это этикетки, а не ящик. Если ссылочные переменные – для вас давно не новость, то все равно аналогия может пригодиться, когда понадобится объяснить кому-нибудь, что такое псевдонимы.

Затем мы обсудим понятия идентичности объектов, значений и псевдонимов. Обнаружится удивительная особенность кортежей: сами они неизменяемы, но их значения могут изменяться. Это подведет нас к вопросу о глубоком и поверхностном копировании. Следующая тема – параметры-ссылки и параметры-функции: проблемы значения изменяемого параметра по умолчанию и безопасной обработки изменяемых аргументов, которые клиенты передают нашим функциям.

Последние разделы главы посвящены сборке мусора, команде `del` и избранным фокусам, которые Python проделывает с неизменяемыми объектами.

Это довольно сухая глава, но рассматриваемые в ней проблемы являются источником многих тонких ошибок в реальных Python-программах.

Что нового в этой главе

Рассматриваемые в этой главе механизмы фундаментальные и стабильные. Достойных специального упоминания изменений в новом издании нет.

В конец раздела «Выбор между == и is» я добавил пример использования `is` для проверки на совпадение с охранным объектом и предупреждение о ненадлежащем употреблении оператора `is`.

Раньше эта глава находилась в части IV, но я решил перенести ее пораньше, потому что она выглядит более уместно в качестве завершения части II «Структуры данных», чем в качестве открывающей тему «Объектно-ориентированные идиомы».

Раздел «Слабые ссылки» из первого издания книги теперь выложен в качестве статьи на сайте *fluentpython.com* (*https://www.fluentpython.com/extra/weak-references/*).

Для начала забудем, что переменная – что-то вроде ящика, в котором хранятся данные.

Переменные – не ящики

В 1997 году я прослушал летний курс по Java в МТИ. Профессор, Линн Андреа Стейн[1], отметила, что стандартная метафора «переменные – это ящики» ведет к непониманию ссылочных переменных в объектно-ориентированных языках. Переменные в Python похожи на ссылочные переменные в Java, поэтому лучше представлять их как этикетки, приклеенные к объектам. Следующий пример и рисунок помогут понять, почему.

В примере 6.1 показано простое взаимодействие, которое невозможно объяснить с помощью метафоры переменных как ящиков. На рис. 6.1 наглядно представлено, почему метафора ящика не годится для Python, тогда как метафора этикетки правильно описывает, как в действительности работают переменные.

Пример 6.1. В переменных `a` и `b` хранятся ссылки на один и тот же список, а не копии списка

```
>>> b = a         ❶
>>> a.append(4)   ❷
>>> b             ❸
[1, 2, 3, 4]      ❹
```

❶ Создать список `[1, 2, 3]` и связать с ним переменную `a`

❷ Связать переменную `b` с тем же значением, на которое ссылается `a`.

❸ Изменить список, на который ссылается `a`, добавив в конец еще один элемент.

❹ Можно наблюдать, как это отразилось на переменной `b`. Если считать `b` ящиком, в котором хранилась копия списка `[1, 2, 3]` из ящика `a`, то такое поведение бессмысленно.

[1] Линн Андреа Стейн – удостоенная наград преподаватель информатики, в настоящее время работает в инженерном колледже Олин (*https://www.olin.edu/bios/lynn-andrea-stein*).

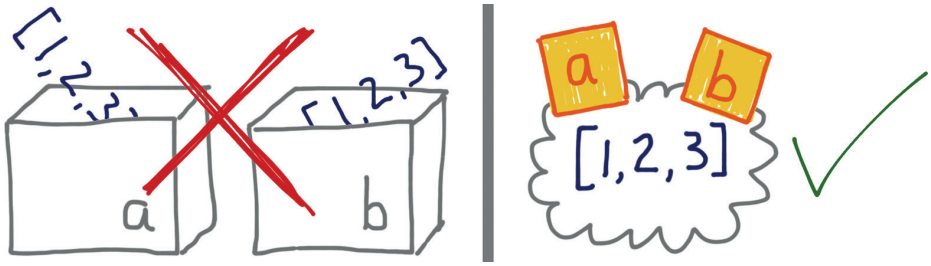


Рис. 6.1. Если представлять себе переменные как ящики, то невозможно понять, как работает присваивание в Python; правильнее считать, что переменные – нечто вроде этикеток, – тогда объяснить пример 6.1 становится проще

Таким образом, предложение `b = a` не копирует содержимое ящика `a` в ящик `b`, а наклеивает метку `b` на объект, уже имеющий метку `a`.

Профессор Стейн также очень аккуратно употребляла слова, говоря о присваивании. Например, рассказывая об объекте seesaw (качели) в программе моделирования, она всегда говорила «переменная *s* присвоена объекту seesaw», а не «объект seesaw присвоен переменной *s*». Имея дело со ссылочными переменными, правильнее говорить, что переменная присвоена объекту, а не наоборот. Ведь объект-то создается раньше присваивания. Пример 6.2 доказывает, что правая часть присваивания вычисляется раньше.

Поскольку употребление глагола «присваивать» (assign) неоднозначно, полезной альтернативой является глагол «связывать» (bind): в Python предложение присваивания `x = …` связывает имя `x` с объектом, находящимся в правой части. И этот объект должен существовать до того, как с ним связывается имя, что доказывает пример 6.2.

Пример 6.2. Переменные связываются с объектами только после создания объектов

```
>>> class Gizmo:
...     def __init__(self):
...         print(f'Gizmo id: {id(self)}')
...
>>> x = Gizmo()
Gizmo id: 4301489152 ❶
>>> y = Gizmo() * 10 ❷
Gizmo id: 4301489432 ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
>>>
>>> dir() ❹
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'x']
```

❶ Вывод `Gizmo id: ...` – побочный эффект создания объекта `Gizmo`.

❷ Умножение объекта `Gizmo` приводит к исключению.

❸ Это доказывает, что второй объект `Gizmo` все-таки был создан еще до попытки выполнить умножение.

❹ Но переменная `y` так и не была создана, потому что исключение произошло тогда, когда вычислялась правая часть.

Для правильного понимания присваивания в Python всегда сначала читайте правую часть, ту, где объект создается или извлекается. Уже после этого переменная в левой части связывается с объектом – как приклеенная к нему этикетка. А о ящиках забудьте.

Поскольку переменные – это просто этикетки, ничто не мешает наклеить на объект несколько этикеток. В этом случае образуются *псевдонимы*. О них и поговорим в следующем разделе.

Тождественность, равенство и псевдонимы

Льюис Кэрролл, литературный псевдоним профессора Чарльза Лутвиджа Доджсона, – не равен проф. Доджсону; это одно и то же лицо. В примере 6.3 эта идея выражена на языке Python.

Пример 6.3. Переменные `charles` и `lewis` ссылаются на один и тот же объект

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles ❶
>>> lewis is charles
True
>>> id(charles), id(lewis) ❷
(4300473992, 4300473992)
>>> lewis['balance'] = 950 ❸
>>> charles
{'name': 'Charles L. Dodgson', 'balance': 950, 'born': 1832}
```

❶ `lewis` – псевдоним `charles`.
❷ Это подтверждают оператор `is` и функция `id`.
❸ Добавление элемента в хеш `lewis` дает тот же результат, что и добавление в хеш `charles`.

Предположим, однако, что некий самозванец – назовем его д-р Александр Педаченко – заявляет, что он и есть Чарльз Л. Доджсон, родившийся в 1832 году. Возможно, он предъявляет такие же документы, но д-р Педаченко и проф. Доджсон – разные лица. Такая ситуация изображена на рис. 6.2.

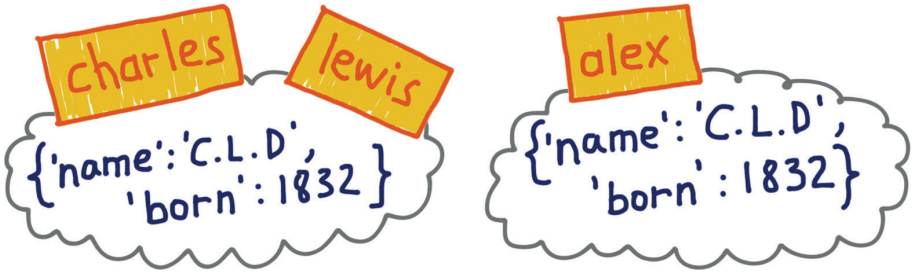


Рис. 6.2. charles и lewis связаны с одним и тем же объектом, alex – с другим объектом, имеющим точно такое же содержимое

В примере 6.4 реализован и протестирован объект `alex`, изображенный на рис. 6.2.

Пример 6.4. `alex` и `charles` равны, но `alex` не совпадает с `charles`

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950} ❶
>>> alex == charles ❷
True
>>> alex is not charles ❸
True
```

❶ Переменная `alex` ссылается на объект, являющийся точной копией объекта, присвоенного переменной `charles`.
❷ При сравнении объекты оказываются равны, поскольку так реализован метод `__eq__` в классе `dict`.
❸ Но это разные объекты. В Python отрицательное сравнение на тождество записывается в виде `a is not b`.

В примере 6.3 иллюстрируются *псевдонимы*. В этом коде `lewis` и `charles` – псевдонимы: две переменные, связанные с одним и тем же объектом. С другой стороны, `alex` не является псевдонимом `charles`: эти переменные связаны с разными объектами. Объекты, связанные с переменными `alex` и `charles`, имеют одно и то же *значение* – то, что сравнивает оператор `==`, – но идентификаторы у них разные.

В разделе 3.1 «Объекты, значения и типы» справочного руководства по языку Python (*https://docs.python.org/3/reference/datamodel.html#objects-values-and-types*) написано:

> У каждого объекта есть идентификатор, тип и значение. Идентификатор объекта после создания не изменяется, можете считать, что это адрес объекта в памяти. Оператор `is` сравнивает идентификаторы двух объектов; функция `id()` возвращает целое число, представляющее идентификатор объекта.

Истинный смысл идентификатора объекта зависит от реализации. В CPython функция `id()` возвращает адрес объекта в памяти, но в другом интерпретаторе это может быть что-то совсем иное. Главное – гарантируется, что идентификатор является уникальной числовой этикеткой и не изменяется в течение всего времени жизни объекта.

На практике мы редко пользуемся функцией `id()`. Проверка на тождество чаще производится с помощью оператора `is`, который сравнивает идентификаторы объектов, поэтому нам вызывать `id()` явно нет необходимости. Далее мы обсудим различия между операторами `is` и `==`.

Технический рецензент Леонардо Рохаэль чаще всего употребляет `id()` во время отладки, когда метод `repr()` возвращает одинаковые представления объектов, но необходимо понять, являются ли две ссылки псевдонимами или ведут на разные объекты. Если ссылки встречаются в разных контекстах – например, в разных кадрах стека, – то использование оператора `is` может не дать желаемого результата.

Выбор между == и is

Оператор `==` сравнивает значения объектов (хранящиеся в них данные), а оператор `is` – их идентификаторы.

При программировании нас обычно интересуют значения, а не идентификаторы, поэтому `==` встречается в Python-программах чаще, чем `is`.

Однако при сравнении переменной с объектом-одиночкой (синглтоном) имеет смысл использовать `is`. Самый типичный случай – проверка того, что переменная связана с объектом `None`. Вот как это рекомендуется делать:

```
x is None
```

А вот как правильно записывать отрицание этого условия:

```
x is not None
```

`None` – самый распространенный синглтон, с которым мы сравниваем с помощью `is`. Охранные объекты – еще один пример такого рода синглтонов. Ниже показан один из способов создания охранного объекта и сравнения с ним:

```
END_OF_DATA = object()
# ... много строчек
def traverse(...):
    # ... еще строчки
    if node is END_OF_DATA:
        return

    # и т. д.
```

Оператор `is` работает быстрее, чем `==`, потому что его невозможно перегрузить, так что интерпретатору не приходится искать и вызывать специальные методы для его вычисления, а само вычисление сводится к сравнению двух целых чисел. Напротив, `a == b` – это синтаксический сахар поверх вызова метода `a.__eq__(b)`. Метод `__eq__`, унаследованный от `object`, сравнивает идентификаторы объектов, поэтому дает тот же результат, что `is`. Но в большинстве встроенных типов метод `__eq__` переопределен в соответствии с семантикой типа, т. е. с учетом значений других атрибутов. Для установления равенства может потребоваться большой объем обработки, например сравнение больших коллекций или глубоко вложенных структур.

Обычно равенство объектов интересует нас больше, чем тождественность. Сравнение с `None` – единственный распространенный случай употребления оператора `is`. В большинстве других ситуаций, с которыми я сталкивался в процессе технического рецензирования кода, `is` употребляется не к месту. Если не уверены, пишите `==`. Обычно это именно то, что нужно, да и для сравнения с `None` тоже работает, правда, не так быстро.

Завершая обсуждение тождественности и равенства, мы покажем, что знаменитый своей неизменяемостью тип `tuple` вовсе не такой несгибаемый, как кажется.

Относительная неизменяемость кортежей

Кортежи, как и большинство коллекций в Python – списки, словари, множества и т. д., – хранят ссылки на объекты[1]. Если элементы, на которые указывают ссылки, изменяемы, то их можно модифицировать, хотя сам кортеж остается неизменяемым. Иными словами, говоря о неизменяемости кортежа, мы име-

[1] С другой стороны, плоские последовательности, например `str`, `bytes` и `array.array`, содержат не ссылки, а сами данные – символы, байты и числа – в непрерывной области памяти.

ем в виду физическое содержимое структуры данных `tuple` (т. е. хранящиеся в ней ссылки), но не объекты, на которые эти ссылки указывают.

В примере 6.5 иллюстрируется ситуация, когда значение кортежа изменяется в результате модификации изменяемого объекта, на который указывает ссылка. Но что никогда не может измениться, так это идентификаторы элементов, хранящихся в кортеже.

Пример 6.5. Кортежи `t1` и `t2` первоначально равны, но после модификации изменяемого объекта, хранящегося в `t1`, они перестают быть равными

```
>>> t1 = (1, 2, [30, 40]) ❶
>>> t2 = (1, 2, [30, 40]) ❷
>>> t1 == t2 ❸
True
>>> id(t1[-1]) ❹
4302515784
>>> t1[-1].append(99) ❺
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1]) ❻
4302515784
>>> t1 == t2 ❼
False
```

❶ `t1` неизменяемый, но `t1[-1]` изменяемый.
❷ Построить кортеж `t2`, элементы которого равны элементам `t1`.
❸ Хотя `t1` и `t2` – разные объекты, они, как и следовало ожидать, равны.
❹ Вывести идентификатор списка в элементе `t1[-1]`.
❺ Модифицировать `t1[-1]` на месте.
❻ Идентификатор объекта `t1[-1]` не изменился, изменилось лишь его значение. `t1` и `t2` теперь не равны.

Эта относительная неизменяемость объясняет загадку в разделе «Головоломка: присваивание A +=» главы 2. По этой же причине некоторые кортежи не являются хешируемыми, как мы видели во врезке «Что значит “хешируемый”»? в главе 2.

Различие между равенством и тождественностью проявляется и при копировании объекта. Копия – это объект, равный исходному, но с другим идентификатором. Однако если объект содержит другие объекты, то следует ли при копировании дублировать также внутренние объекты или можно оставить их разделяемыми? Единственно правильного ответа на этот вопрос не существует. Читайте дальше.

По умолчанию копирование поверхностное

Простейший способ скопировать список (как и большинство встроенных изменяемых коллекций) – воспользоваться встроенным конструктором самого типа, например:

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1) ❶
```

```
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1 ❷
True
>>> l2 is l1 ❸
False
```

❶ `list(l1)` создает копию `l1`.
❷ Копии равны…
❸ … но ссылаются на разные объекты.

Для списков и других изменяемых последовательностей присваивание `l2 = l1[:]` также создает копию.

Однако при использовании конструктора и оператора `[:]` создается *поверхностная копия* (т. е. дублируется только самый внешний контейнер, который заполняется ссылками на те же элементы, что хранятся в исходном контейнере). Это экономит память и не создает проблем, если все элементы неизменяемые. Однако при наличии изменяемых элементов можно столкнуться с неприятными сюрпризами.

В примере 6.6 мы создаем поверхностную копию списка, который содержит другой список и кортеж, а затем производим изменения и смотрим, как они отразились на объектах, на которые указывают ссылки.

Если ваш компьютер подключен к сети, рекомендую понаблюдать за интерактивной анимацией примера 6.6 на сайте Online Python Tutor (*http://www.pythontutor.com/*). Во время работы над этой главой прямая ссылка на пример, подготовленный для *pythontutor.com*, работала ненадежно, но сам инструмент замечательный, поэтому время, потраченное на копирование кода на сайт, будет потрачено не зря.

Пример 6.6. Создание поверхностной копии списка, содержащего другой список; скопируйте этот код на сайт Online Python Tutor, чтобы увидеть его анимацию

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)     ❶
l1.append(100)    ❷
l1[1].remove(55)  ❸
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22] ❹
l2[2] += (10, 11) ❺
print('l1:', l1)
print('l2:', l2)
```

❶ `l2` – поверхностная копия `l1`. Это состояние изображено на рис. 6.3.
❷ Добавление `100` в `l1` не отражается на `l2`.
❸ Здесь мы удаляем `55` из внутреннего списка `l1[1]`. Это отражается на `l2`, потому что объект `l2[1]` связан с тем же списком, что `l1[1]`.
❹ Для изменяемого объекта, в частности списка, на который ссылается `l2[1]`, оператор `+=` изменяет список на месте. Это изменение отражается на `l1[1]`, т. к. это псевдоним `l2[1]`.
❺ Для кортежа оператор `+=` создает новый кортеж и переприязывает к нему переменную `l2[2]`. Это то же самое, что присваивание `l2[2] = l2[2] + (10, 11)`.

Отметим, что кортежи в последней позиции списков `l1` и `l2` уже не являются одним и тем же объектом. См. рис. 6.4.

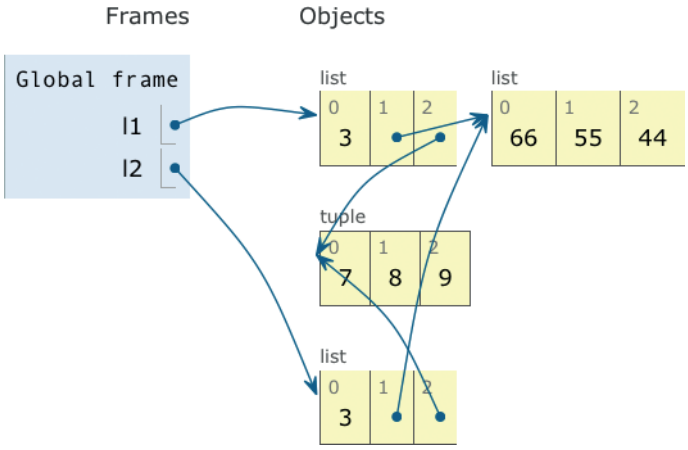


Рис. 6.3. Состояние программы сразу после присваивания `l2 = list(l1)` в примере 6.6. `l1` и `l2` ссылаются на разные списки, но эти списки разделяют ссылки на один и тот же объект внутреннего списка `[66, 55, 44]` и кортеж `(7, 8, 9)` (рисунок построен сайтом Online Python Tutor)

Результат работы примера 6.6 показан в примере 6.7, а конечное состояние объектов – на рис. 6.4.

Пример 6.7. Результат работы примера 6.6

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

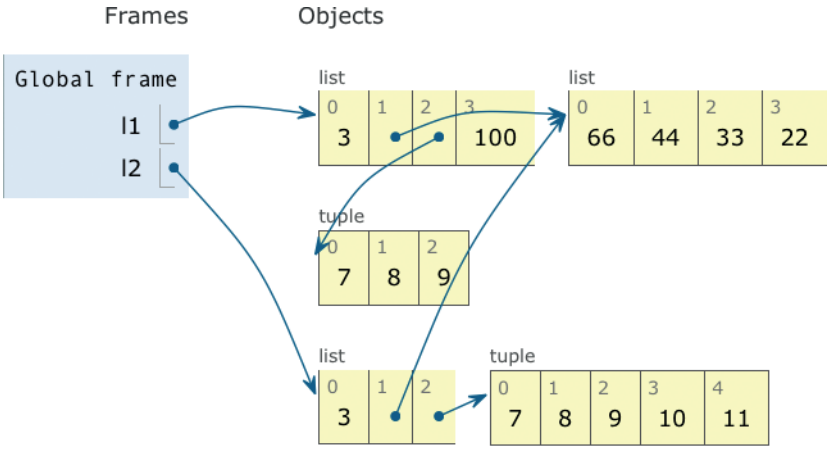


Рис. 6.4. Конечное состояние `l1` и `l2`: они по-прежнему разделяют ссылки на один и тот же объект списка, который теперь содержит `[66, 44, 33, 22]`, но в результате операции `l2[2] += (10, 11)` был создан новый кортеж `(7, 8, 9, 10, 11)`, не связанный с кортежем `(7, 8, 9)`, на который ссылается элемент `l1[2]` (рисунок построен сайтом Online Python Tutor)

Теперь должно быть понятно, что создать поверхностную копию легко, но это не всегда то, что нам нужно. В следующем разделе мы обсудим создание глубоких копий.

Глубокое и поверхностное копирование произвольных объектов

Не всегда поверхностное копирование является проблемой, но иногда требуется получить глубокую копию (когда копия не разделяет с оригиналом ссылки на внутренние объекты). В модуле `copy` имеются функции `deepcopy` и `copy`, которые возвращают соответственно глубокие и поверхностные копии произвольных объектов.

Для иллюстрации работы `copy()` и `deepcopy()` в примере 6.8 определен простой класс `Bus`, представляющий школьный автобус, который по ходу маршрута подбирает и высаживает пассажиров.

Пример 6.8. Автобус подбирает и высаживает пассажиров

```
class Bus:
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)
```

Далее в интерактивном примере 6.9 мы создадим объект класса `Bus` (`bus1`) и два его клона: поверхностную копию (`bus2`) и глубокую копию (`bus3`) – и понаблюдаем за тем, что происходит, когда `bus1` высаживает школьника.

Пример 6.9. Сравнение `copy` и `deepcopy`

```
>>> import copy
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
>>> bus2 = copy.copy(bus1)
>>> bus3 = copy.deepcopy(bus1)
>>> id(bus1), id(bus2), id(bus3)
(4301498296, 4301499416, 4301499752) ❶
>>> bus1.drop('Bill')
>>> bus2.passengers
['Alice', 'Claire', 'David'] ❷
>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)
(4302658568, 4302658568, 4302657800) ❸
>>> bus3.passengers
['Alice', 'Bill', 'Claire', 'David'] ❹
```

❶ Используя `copy` и `deepcopy`, мы создаем три разных объекта `Bus`.
❷ После высадки `'Bill'` из автобуса `bus1` он исчезает и из `bus2`.

❸ Инспекция атрибута `passengers` показывает, что `bus1` и `bus2` разделяют один и тот же объект списка, т. к. `bus2` – поверхностная копия `bus1`.
❹ `bus3` – глубокая копия `bus1`, поэтому ее атрибут `passengers` ссылается на другой список.

Отметим, что в общем случае создание глубокой копии – дело не простое. Между объектами могут существовать циклические ссылки, из-за которых наивный алгоритм попадет в бесконечный цикл. Для корректной обработки циклических ссылок функция `deepcopy` запоминает, какие объекты она уже копировала. Это продемонстрировано в примере 6.10.

Пример 6.10. Циклические ссылки: `b` ссылается на `a`, а затем добавляется в конец `a`; тем не менее `deepcopy` справляется с копированием `a`

```
>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
>>> c = deepcopy(a)
>>> c
[10, 20, [[...], 30]]
```

Кроме того, в некоторых случаях глубокое копирование может оказаться слишком глубоким. Например, объекты могут ссылаться на внешние ресурсы или на синглтоны, которые копировать не следует. Поведением функций `copy` и `deepcopy` можно управлять, реализовав специальные методы `__copy__()` и `__deepcopy__()`, как описано в документации по модулю `copy` (*http://docs.python.org/3/library/copy.html*).

Разделение ссылок на объекты посредством псевдонимов объясняет также механизм передачи параметров в Python и решает проблему использования изменяемых типов для параметров по умолчанию. Эти вопросы мы рассмотрим далее.

Параметры функций как ссылки

Единственный способ передачи параметров в Python – *вызов по соиспользованию* (call by sharing). Он применяется в большинстве объектно-ориентированных языков, в том числе JavaScript, Ruby, и Java (это относится к ссылочным типам Java, параметры примитивных типов передаются по значению). Вызов по соиспользованию означает, что каждый формальный параметр функции получает копию ссылки на фактический аргумент. Иначе говоря, внутри функции параметры становятся псевдонимами фактических аргументов.

В результате функция получает возможность модифицировать любой изменяемый объект, переданный в качестве параметра, но не может заменить объект другим, не тождественным ему. В примере 6.11 показана простая функция, применяющая оператор `+=` к одному из своих параметров. Результат зависит от того, что передано в качестве фактического аргумента: число, список или кортеж.

Пример 6.11. Функция может модифицировать любой переданный ей изменяемый объект

```
>>> def f(a, b):
...     a += b
...     return a
...
>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y ❶
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b ❷
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
>>> u = (30, 40)
>>> f(t, u) ❸
(10, 20, 30, 40)
>>> t, u
((10, 20), (30, 40))
```

❶ Число `x` не изменилось.
❷ Список `a` изменился.
❸ Кортеж `t` не изменился.

С параметрами функций связан также еще один вопрос: что бывает, когда значение по умолчанию имеет изменяемый тип?

Значения по умолчанию изменяемого типа: неудачная мысль

Необязательные параметры, имеющие значения по умолчанию, – замечательная возможность, которую можно использовать в определениях функций для обеспечения обратной совместимости API. Однако не следует использовать в качестве значений по умолчанию изменяемые объекты.

Для иллюстрации возникающей проблемы мы в примере 6.12 взяли класс `Bus` из примера 6.8 и изменили в нем метод `__init__`, получив новый класс `HauntedBus`. Но решили поумничать и вместо значения по умолчанию `passengers=None` задали `passengers=[]`, избавившись тем самым от предложения `if` в предыдущем варианте `__init__`. Такое «умничанье» приводит к беде.

Пример 6.12. Простой класс, иллюстрирующий опасности изменяемых значений по умолчанию

```
class HauntedBus:
    """Автобус, облюбованный пассажирами-призраками"""

    def __init__(self, passengers=[]): ❶
        self.passengers = passengers ❷

    def pick(self, name):
        self.passengers.append(name) ❸
```

```
    def drop(self, name):
        self.passengers.remove(name)
```

❶ Если аргумент `passengers` не передан, то этот параметр связывается с объектом списка по умолчанию, который первоначально пуст.
❷ В результате этого присваивания `self.passengers` становится псевдонимом `passengers`, который сам является псевдонимом списка по умолчанию, если аргумент `passengers` не передан.
❸ Применяя методы `.remove()` и `.append()` к `self.passengers`, мы на самом деле изменяем список по умолчанию, который является атрибутом объекта-функции.

В примере 6.13 показано потустороннее поведение объекта `HauntedBus`.

Пример 6.13. Автобусы, облюбованные пассажирами-призраками

```
>>> bus1 = HauntedBus(['Alice', 'Bill']) ❶
>>> bus1.passengers
['Alice', 'Bill']
>>> bus1.pick('Charlie')
>>> bus1.drop('Alice')
>>> bus1.passengers ❷
['Bill', 'Charlie']
>>> bus2 = HauntedBus() ❸
>>> bus2.pick('Carrie')
>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus() ❹
>>> bus3.passengers
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers ❺
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers ❻
True
>>> bus1.passengers ❽
['Bill', 'Charlie']
```

❶ Вначале `bus1` содержит список из двух пассажиров.
❷ Пока все хорошо: `bus1` не таит никаких сюрпризов.
❸ `bus2` вначале пуст, поэтому атрибуту `self.passengers` присвоен пустой список по умолчанию.
❹ `bus3` также вначале пуст, `self.passengers` – снова список по умолчанию.
❺ Список по умолчанию уже не пуст!
❻ Теперь `Dave`, севший в автобус `bus3`, оказался и в `bus2`.
❼ Проблема: `bus2.passengers` и `bus3.passengers` ссылаются на один и тот же список.
❽ Но `bus1.passengers` – другой список.

Проблема в том, что все экземпляры `HauntedBus`, конструктору которых не был явно передан список пассажиров, разделяют один и тот же список по умолчанию.

Это тонкая ошибка. Из примера 6.13 видно, что когда объект `HauntedBus` инициализируется списком пассажиров, он работает правильно. Странности начинаются, когда `HauntedBus` вначале пуст, потому что в этом случае `self.passengers`

оказывается псевдонимом значения по умолчанию для параметра `passengers`. Беда в том, что любое значение по умолчанию вычисляется в момент определения функции, т. е. обычно на этапе загрузки модуля, после чего значения по умолчанию становятся атрибутами объекта-функции. Так что если значение по умолчанию – изменяемый объект и вы его изменили, то изменение отразится и на всех последующих вызовах функции.

Если после выполнения кода из примера 6.13 проинспектировать объект `HauntedBus.__init__`, то мы обнаружим школьников-призраков в его атрибуте `__defaults__`:

```
>>> dir(HauntedBus.__init__) # doctest: +ELLIPSIS
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'],)
```

Наконец, можно убедиться, что `bus2.passengers` – псевдоним первого элемента атрибута `HauntedBus.__init__.__defaults__`:

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
True
```

Описанная проблема и есть причина того, почему для параметров, принимающих изменяемые значения, часто по умолчанию задается значение `None`. В примере 6.8 `__init__` проверяет, верно ли, что аргумент `passengers` совпадает с `None`, и, если это так, присваивает атрибуту `self.passengers` вновь созданный пустой список. Если `passengers` не совпадает с `None`, то правильное решение заключается в том, чтобы связать копию этого атрибута с `self.passengers`. В следующем разделе объясняется, почему следует предпочесть копирование аргумента.

Защитное программирование при наличии изменяемых параметров

При написании функции, принимающей изменяемый параметр, нужно тщательно обдумать, ожидает ли вызывающая сторона, что переданный аргумент может быть изменен.

Например, если функция принимает словарь и должна модифицировать его в процессе обработки, должен ли этот побочный эффект быть виден вне самой функции? Ответ зависит от контекста. Так или иначе, необходимо согласовать предположения автора функции и вызывающей программы.

Приведем еще один, последний, пример автобуса – класс `TwilightBus`, который нарушает ожидания, разделяя список пассажиров со своими клиентами. Прежде чем переходить к реализации, посмотрите, как работает класс `TwilightBus` с точки зрения его клиента.

Пример 6.14. Пассажиры, вышедшие из автобуса `TwilightBus`, бесследно исчезают

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat'] ❶
>>> bus = TwilightBus(basketball_team) ❷
>>> bus.drop('Tina') ❸
>>> bus.drop('Pat')
>>> basketball_team
['Sue', 'Maya', 'Diana'] ❹
```

❶ В списке `basketball_team` пять школьников.
❷ `TwilightBus` везет всю баскетбольную команду.
❸ Из автобуса `bus` вышел сначала один школьник, за ним второй.
❹ Вышедшие пассажиры исчезли из баскетбольной команды!

Класс `TwilightBus` нарушает «принцип наименьшего удивления»[1] – одну из рекомендаций по проектированию интерфейсов. Поистине удивительно, что стоит школьнику выйти из автобуса, как он исчезает из состава баскетбольной команды.

В примере 6.15 приведена реализация класса `TwilightBus` и объяснена причина проблемы.

Пример 6.15. Простой класс, иллюстрирующий опасности, которыми чревато изменение полученных аргументов

```
class TwilightBus:
    """Автобус, из которого бесследно исчезают пассажиры"""

    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = [] ❶
        else:
            self.passengers = passengers ❷

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name) ❸
```

❶ Здесь мы честно создаем пустой список, когда `passengers` совпадает с `None`.
❷ Но в результате этого присваивания `self.passengers` становится псевдонимом параметра `passengers`, который сам является псевдонимом фактического аргумента, переданного методу `__init__` (т. е. `basketball_team` в примере 6.14).
❸ Применяя методы `.remove()` и `.append()` к `self.passengers`, мы в действительности изменяем исходный список, переданный конструктору в качестве аргумента.

Проблема здесь в том, что в объекте `bus` создается псевдоним списка, переданного конструктору. А надо бы хранить собственный список пассажиров. Исправить ошибку просто: в методе `__init__` атрибут `self.passengers` следует инициализировать копией параметра `passengers`, если тот задан, как и было сделано в примере 6.8.

```
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers) ❶
```

❶ Создать копию списка `passengers` или преобразовать его в тип `list`, если параметр имеет другой тип.

[1] См. статью «Principle of least astonishment» (*https://en.wikipedia.org/wiki/Principle_of_least_astonishment*) в англоязычной части Википедии.

Вот теперь внутренние операции со списком пассажиров никак не влияют на аргумент, переданный конструктору автобуса. Заодно это решение оказывается и более гибким: аргумент, переданный в качестве параметра `passengers`, может быть кортежем или любым другим итерируемым объектом, например множеством или даже результатом запроса к базе данных, поскольку конструктор класса `list` принимает любой итерируемый объект. Так как мы сами создали список, с которым будем работать, то гарантируется, что он поддерживает операции `.remove()` и `.append()`, используемые в методах `.pick()` и `.drop()`.

Если метод специально не предназначен для изменения объекта, полученного в качестве аргумента, то следует дважды подумать, перед тем как создавать псевдоним аргумента, просто присваивая его атрибуту экземпляра в своем классе. Если сомневаетесь, делайте копию. Клиенты обычно будут только рады. Конечно, создание копии обходится не бесплатно: потребляется процессорное время и память. Однако API, вызывающий тонкие ошибки, – обычно куда большая проблема, чем небольшое замедление или немного повышенное потребление ресурсов.

Теперь поговорим об одном из самых плохо понимаемых предложений Python: `del`.

DEL И СБОРКА МУСОРА

> Объекты никогда не уничтожаются явно, однако, оказавшись недоступными, они могут стать жертвой сборщика мусора.
>
> – «Модель данных», глава справочного руководства по языку Python

Первая странность `del` – то, что это не функция, а предложение языка. Мы пишем `del x`, а не `del(x)`, хотя вторая форма тоже работает, но только потому, что в Python выражения `x` и `(x)` обычно означают одно и то же.

Вторая странность – то, что предложение `del` удаляет ссылки, а не объекты. Сборщик мусора в Python может удалить объект из памяти в качестве побочного результата `del`, если это была последняя ссылка на объект. Связывание переменной с другим объектом также может обнулить количество ссылок на объект, что приведет к его уничтожению.

```
>>> a = [1, 2] ❶
>>> b = a      ❷
>>> del a      ❸
>>> b          ❹
[1, 2]
>>> b = [3]    ❺
```

❶ Создать объект `[1, 2]` и связать с ним `a`.
❷ Связать `b` с тем же самым объектом `[1, 2]`.
❸ Удалить ссылку `a`.
❹ Объект `[1, 2]` остался на месте, потому что `b` по-прежнему указывает на него.
❺ Если связать `b` с другим объектом, то последняя оставшаяся ссылка на `[1, 2]` будет удалена. Теперь сборщик мусора может удалить сам объект.

Существует специальный метод `__del__`, но он не приводит к уничтожению экземпляра, и вы не должны вызывать его самостоятельно. Метод `__del__` вызывается интерпретатором Python непосредственно перед уничтожением объекта, давая ему возможность освободить внешние ресурсы. Вам редко придется реализовывать метод `__del__` в своем коде, но тем не менее некоторые начинающие программисты тратят время на его написание безо всяких на то причин. Правильно написать метод `__del__` довольно сложно. Он документирован в главе «Модель данных» справочного руководства по языку Python (*https://docs.python.org/3/reference/datamodel.html# object.__del__*).

В CPython основной алгоритм сборки мусора основан на подсчете ссылок. В каждом объекте хранится счетчик указывающих на него ссылок – *refcount*. Как только этот счетчик обратится в нуль, объект сразу же уничтожается: CPython вызывает метод `__del__` объекта (если он определен), а затем освобождает выделенную ему память. В CPython 2.0 был добавлен алгоритм сборки мусора, основанный на поколениях, который обнаруживает группы объектов, ссылающихся друг на друга и образующих замкнутую группу. Такие объекты могут оказаться недостижимыми, хотя в каждом из них счетчик ссылок больше нуля. В других реализациях Python применяются более сложные сборщики мусора, не опирающиеся на подсчет ссылок, а это означает, что метод `__del__` может вызываться не сразу после того, как на объект не осталось ссылок. См. статью A. Jesse Jiryu Davis «PyPy, Garbage Collection, and a Deadlock» (*https://emptysqua.re/blog/pypy-garbage-collection-and-a-deadlock/*), в которой обсуждается правильное и неправильное использование метода `__del__`.

Для демонстрации завершения жизни объекта в примере 6.16 используется функция `weakref.finalize`, которая регистрирует функцию обратного вызова, вызываемую перед уничтожением объекта.

Пример 6.16. Наблюдение за гибелью объекта, на который не осталось ссылок

```
>>> import weakref
>>> s1 = {1, 2, 3}
>>> s2 = s1 ❶
>>> def bye(): ❷
...     print('...like tears in the rain.')
...
>>> ender = weakref.finalize(s1, bye) ❸
>>> ender.alive ❹
True
>>> del s1
>>> ender.alive ❺
True
>>> s2 = 'spam' ❻
...like tears in the rain.
>>> ender.alive
False
```

❶ `s1` и `s2` – псевдонимы, ссылающиеся на одно и то же множество `{1, 2, 3}`.

❷ Эта функция не должна быть связанным методом уничтожаемого объекта, иначе она будет хранить ссылку на него.

❸ Регистрируем обратный вызов `bye` объекта, на который ссылается `s1`.
❹ Атрибут `.alive` равен `True`, перед тем как вызвана функция, зарегистрированная `finalize`.
❺ Как было сказано, `del` удаляет не объект, а только ссылку на него.
❻ После перепривязки последней ссылки, `s2`, объект `{1, 2, 3}` оказывается недоступен. Он уничтожается, вызывается функция `bye`, и атрибут `ender.alive` становится равен `False`.

Смысл примера 6.16 – ясно показать, что предложение `del` не удаляет объекты, хотя объекты могут быть удалены из-за того, что после выполнения `del` оказываются недоступны.

Быть может, вам не понятно, почему в примере 6.16 был уничтожен объект `{1, 2, 3}`. Ведь ссылка `s1` была передана функции `finalize`, которая должна была бы удержать ее, чтобы следить за объектом и вызвать функцию обратного вызова. Это работает, потому что `finalize` удерживает *слабую ссылку* на объект `{1, 2, 3}`. Слабые ссылки на объект не увеличивают счетчик ссылок. Таким образом, слабая ссылка не препятствует уничтожению объекта ссылки сборщиком мусора. Слабые ссылки полезны для кеширования, поскольку мы не хотим, чтобы кешированный объект оставался жив только потому, что на него ссылается сам кеш.

Слабые ссылки – очень специальная тема. Поэтому я решил опустить посвященный им раздел во втором издании. Но вы можете найти его на сайте *fluentpython.com* в статье «Weak References» (*https://www.fluentpython.com/extra/weak-references/*).

Как Python хитрит с неизменяемыми объектами

Вы можете спокойно пропустить этот раздел. В нем обсуждаются некоторые детали реализации, которые *пользователям* Python не особенно интересны и могут быть неприменимы к другим реализациям Python и даже к будущим версиям CPython. Тем не менее, экспериментируя с псевдонимами и копированием, иногда можно наткнуться на следы этих трюков, поэтому мне показалось, что о них стоит сказать пару слов.

Я удивился, узнав, что для кортежа `t` конструкция `t[:]` не создает копию, а возвращает ссылку на сам объект. Ссылку на исходный кортеж мы получаем также, написав `tuple(t)`[1]. Это доказывает пример 6.17.

Пример 6.17. Кортеж, инициализированный другим кортежем, в точности совпадает с исходным

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1 ❶
True
>>> t3 = t1[:]
```

[1] Это поведение четко документировано. Набрав `help(tuple)` в оболочке Python, читаем: «Если аргумент является кортежем, то возвращается исходный объект». А я-то, садясь за написание этой книги, думал, что знаю о кортежах все.

```
>>> t3 is t1 ❷
True
```

❶ t1 и t2 связаны с одним и тем же объектом.
❷ И t3 тоже.

Такое же поведение свойственно экземплярам классов str, bytes и frozenset. Отметим, что frozenset – не последовательность, поэтому, когда fs является объектом frozenset, конструкция fs[:] не работает. Но fs.copy() дает точно такой же эффект: обманывает нас и возвращает ссылку на тот же объект, а вовсе не на его копию. См. пример 6.18[1].

Пример 6.18. Строковые литералы могут создавать разделяемые объекты

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3) ❶
>>> t3 is t1 ❷
False
>>> s1 = 'ABC'
>>> s2 = 'ABC' ❸
>>> s2 is s1 ❹
True
```

❶ Создание нового кортежа с нуля.
❷ t1 и t3 равны, но не тождественны.
❸ Создание второй строки str с нуля.
❹ Сюрприз: s1 и s2 ссылаются на один и тот же объект str!

Разделение строковых литералов – это техника оптимизации, называемая *интернированием*. В CPython тот же прием используется для небольших целых чисел, чтобы избежать ненужного дублирования «популярных» чисел, например 0, 1, –1 и т. д. Отметим, что CPython не интернирует все строки и целые числа подряд, а критерии, которыми он руководствуется, остаются недокументированной деталью реализации.

Никогда не полагайтесь на интернирование объектов str и int! Для сравнения на равенство используйте только оператор ==, а не is. Интернирование предназначено исключительно для внутренних нужд интерпретатора.

Трюки, обсуждаемые в этом разделе, в том числе поведение метода frozenset.copy(), – это «ложь во спасение»: они экономят память и ускоряют работу интерпретатора. Не думайте о них, никаких хлопот они не доставят, потому что относятся только к неизменяемым объектам. Пожалуй, наилучшее применение этим мелочам – пари со знакомыми питонистами[2].

[1] Невинную ложь – тот факт, что метод copy ничего не копирует, – можно объяснить совместимостью интерфейсов: при этом класс frozenset оказывается лучше совместим с set. Как бы то ни было, конечному пользователю безразлично, являются два идентичных неизменяемых объекта одним и тем же объектом или разными.

[2] Жестокое использование этой информации – задать вопрос кандидату на собеседовании или включить его в экзамен на получение сертификата. Есть множество более важных и полезных вещей, позволяющих судить о знании Python.

Резюме

У каждого объекта в Python есть идентификатор, тип и значение. И только значение объекта может изменяться со временем[1].

Если две переменные ссылаются на неизменяемые объекты, имеющие равные значения (`a == b` принимает значение `True`), то на практике редко бывает важно, ссылаются они на копии или являются псевдонимами, – за одним исключением. Это исключение составляют неизменяемые коллекции, например кортежи и объекты `frozenset`: если неизменяемая коллекция содержит ссылки на изменяемые объекты, то ее значение может измениться при изменении значения одного из ее элементов. На практике такая ситуация встречается нечасто. Но идентификаторы объектов, хранящихся в неизменяемой коллекции, не изменяются ни при каких обстоятельствах. Классу `frozenset` эта проблема не свойственна, потому что в нем могут храниться только хешируемые элементы, а значение хешируемого объекта, по определению, никогда не изменяется.

Из того, что в переменных хранятся ссылки, вытекает ряд практических следствий.

- Простое присваивание не создает копий.
- Составное присваивание (операторы `+=`, `*=` и т. п.) создает новый объект, если переменная в левой части связана с неизменяемым объектом, а изменяемый объект может быть модифицирован на месте.
- Присваивание нового значения существующей переменной не изменяет объект, с которым она была связана ранее. Это называется перепривязкой: переменная просто связывается с другим объектом. Если в этой переменной хранилась последняя ссылка на предыдущий объект, то этот объект убирается в мусор.
- Параметры функций передаются как псевдонимы, т. е. функция может модифицировать любой изменяемый объект, переданный ей в качестве аргумента. Этому невозможно воспрепятствовать, разве что создать локальную копию или использовать неизменяемые объекты (т. е. передавать кортеж вместо списка).
- Использовать изменяемые объекты в качестве значений параметров функции по умолчанию опасно, потому что если изменить параметр на месте, то изменится значение по умолчанию, и это скажется на всех последующих вызовах функции с параметром по умолчанию.

В CPython объект уничтожается, как только число ссылок на него станет равно нулю. Объекты также могут уничтожаться, если образуют группу с циклическими ссылками друг на друга, и ни на один объект группы нет других – внешних – ссылок.

В некоторых ситуациях полезно иметь ссылку, которая сама по себе не удерживает объект «в мире живых». Примером может служить класс, желающий отслеживать все свои экземпляры. Это можно сделать с помощью слабых ссы-

[1] На самом деле тип объекта тоже можно изменить, просто присвоив другой класс его атрибуту `__class__`, но это неприкрытое зло, и я жалею, что написал эту сноску.

лок – низкоуровневого механизма, на базе которого построены более полезные коллекции `WeakValueDictionary`, `WeakKeyDictionary`, `WeakSet` и функция `finalize` – все из модуля `weakref`. Дополнительные сведения по этому вопросу см. в статье «Weak References» (*https://www.fluentpython.com/extra/weak-references/*).

Дополнительная литература

Глава «Модель данных» (*https://docs.python.org/3/reference/datamodel.html*) справочного руководства по языку Python начинается с объяснения того, что такое идентификаторы и значения объектов.

Уэсли Чан, автор серии книг *Core Python*, показал на конференции EuroPython 2011 презентацию «Understanding Python's Memory Model, Mutability, and Methods» (*https://www.youtube.com/watch?v=HHFCFJSPWrI&feature=youtu.be*), в которой затрагиваются не только темы этой главы, но и использование специальных методов.

Дуг Хеллманн написал статьи «copy – Duplicate Objects» (*http://pymotw.com/2/copy/*) и «weakref – Garbage-Collectable References to Objects» (*http://pymotw.com/2/weakref/*), где рассматриваются некоторые из обсуждавшихся в этой главе вопросов.

Дополнительные сведения об основанном на поколениях сборщике мусора можно найти в документации по модулю `gc` (*https://docs.python.org/3/library/gc.html*), которая начинается фразой: «Этот модуль предоставляет интерфейс к факультативному сборщику мусора». Слово «факультативный» в этом контексте может вызвать удивление, но в главе «Модель данных» (*https://docs.python.org/3/reference/datamodel.html*) также утверждается:

> Реализации разрешено откладывать сборку мусора и даже не производить ее вовсе; как именно реализована сборка мусора – вопрос качества реализации, главное условие – чтобы не уничтожался ни один объект, который все еще достижим.

Пабло Галиндо представил развернутое описание сборщика мусора в Python в разделе «Design of CPython's Garbage Collector» (*https://devguide.python.org/garbage_collector/*) руководства по Python для разработчиков, ориентированное на новых и опытных соразработчиков CPython.

В версии CPython 3 сборщик мусора усовершенствован в части обработки объектов с методом `__del__`, это описано в документе PEP 442 «Safe object finalization» (*https://peps.python.org/pep-0442/*).

В Википедии имеется статья об интернировании строк (*https://en.wikipedia.org/wiki/String_interning*), в которой описывается применение этой техники в разных языках, включая Python.

В Википедии также имеется статья о песне Льюиса Кэрролла «Haddocks' Eyes» (Пуговки для сюртуков) (*https://en.wikipedia.org/wiki/Haddocks%27_Eyes*), которую я вынес в эпиграф к этой главе. Редакторы Википедии пишут, что эти стихи использовались в работах по логике и философии, «чтобы прояснить символический статус понятия *имени*: имя – это идентифицирующий маркер, который можно связать с чем угодно, в т. ч. с другим именем, и тем самым организовать различные уровни представления в виде символов».

Поговорим

Равное отношение ко всем объектам

Прежде чем открыть для себя Python, я изучал Java. Оператор `==` в Java всегда оставлял у меня чувство неудовлетворенности. Программист обычно интересуется равенством, а не тождественностью, но для объектов (в отличие от примитивных типов) оператор == сравнивает ссылки, а не значения объектов. Даже такую базовую вещь, как сравнение строк, Java заставляет делать с применением метода `.equals`. Но и в этом случае есть подвох: если при вычислении выражения `a.equals(b)` окажется, что `a` равно `null`, то возникнет исключение из-за нулевого указателя. Проектировщики Java сочли необходимым переопределить для строк оператор `+`, так почему же не пошли дальше и не переопределили также оператор `==`?

В Python это сделано правильно. Оператор `==` сравнивает значения объектов, а оператор `is` – ссылки. И поскольку в Python имеется механизм перегрузки операторов, то `==` разумно работает для всех объектов из стандартной библиотеки, включая `None`, каковой является обычным объектом, в отличие от `null` в Java.

И разумеется, можно определить метод `__eq__` в собственных классах, самостоятельно решив, что должен означать для них оператор `==`. Если метод `__eq__` не переопределен, то он наследуется от `object` и сравнивает идентификаторы объектов, так что все объекты пользовательского класса по умолчанию считаются различными.

Вот такие вещи побудили меня перейти с Java на Python, после того как в один прекрасный день в сентябре 1998 года я прочел «Учебное пособие по Python».

Изменяемость

Эта глава была бы излишней, если бы все объекты в Python были неизменяемы. Когда имеешь дело с неизменяемым объектом, не важно, хранятся ли в переменных сами объекты или ссылки на разделяемые объекты. Если `a == b` истинно и ни тот, ни другой объект не может измениться, то они вполне могут быть одним и тем же объектом. Вот поэтому интернирование строк и безопасно. Тождественность объектов становится важна, только если объекты изменяемы.

В «чистом» функциональном программировании все данные неизменяемы: при добавлении элемента в коллекцию создается новая коллекция. Например, Elixir – практичный функциональный язык, в котором все встроенные типы, включая списки, неизменяемы.

Но Python – не функциональный язык программирования и уж тем более не чистый. Экземпляры пользовательских классов в Python по умолчанию изменяемы – как и в большинстве объектно-ориентированных языков. Если требуется создавать неизменяемые объекты, то следует проявлять особую осторожность. Каждый атрибут такого объекта тоже должен быть неизменяемым, иначе получится нечто, аналогичное кортежу: хотя с точки зрения идентификаторов объектов кортеж неизменяемый, его значение может измениться, если в нем хранятся изменяемые объекты.

Изменяемые объекты – также основная причина, из-за которой так трудно написать корректную многопоточную программу: если потоки изменяют

объекты, не заботясь о синхронизации, то данные будут повреждены. С другой стороны, если синхронизации слишком много, возникают взаимоблокировки. Язык и платформа Erlang (включающая и Elixir) проектировались, чтобы обеспечить максимально долгое непрерывное функционирование в сильно конкурентных распределенных приложениях, например телекоммуникационных коммутаторах. Естественно, что по умолчанию данные в них неизменяемые.

Уничтожение объектов и сборка мусора

В Python нет механизма прямого уничтожения объекта, и это не упущение, а великое благо: если бы можно было уничтожить объект в любой момент, что стало бы с указывающими на него сильными ссылками?

В CPython сборка мусора основана главным образом на механизме подсчета ссылок; он легко реализуется, но ведет к утечке памяти при наличии циклических ссылок. Поэтому в версии 2.0 (октябрь 2000) был реализован сборщик мусора на основе поколений, который умеет уничтожать группы объектов, связанных только циклическими ссылками и недостижимых извне.

Но подсчет ссылок по-прежнему остается основным механизмом и приводит к немедленному уничтожению объектов, на которые не осталось ссылок. Это означает, что в CPython – по крайней мере, сейчас – безопасно такое предложение:

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

Этот код безопасен, потому что счетчик ссылок на объект файла окажется равен нулю после возврата из метода `write`, и Python немедленно закроет файл, перед тем как уничтожить объект, представляющий его в памяти. Однако в Jython или IronPython эта строка небезопасна, т. к. они пользуются более сложными сборщиками мусора в объемлющей среде выполнения (Java VM и .NET CLR соответственно), которые не опираются на подсчет ссылок и могут отложить уничтожение объекта и закрытие файла на неопределенное время. Поэтому в любом случае и, в частности, в CPython рекомендуется явно закрывать файл, а самый надежный способ сделать это – воспользоваться предложением `with`, которое гарантирует закрытие файла, даже если, пока он был открыт, произошло исключение. С использованием `with` показанный выше фрагмент можно записать так:

```
with open('test.txt', 'wt', encoding='utf-8') as fp:
    fp.write('1, 2, 3')
```

Если вас заинтересовала тема сборщиков мусора, можете почитать статью Томаса Перла «Python Garbage Collector Implementations: CPython, PyPy and GaS» (*https://thp.io/2012/python-gc/python_gc_final_2012-01-22.pdf*), из которой я узнал о безопасности `open().write()` в CPython.

Передача параметров: вызов по соиспользованию

Популярным объяснением механизма передачи параметров в Python является фраза: «Параметры передаются по значению, но значениями являются ссылки». Нельзя сказать, что это неверно, но вводит в заблуждение, потому что в более старых языках наиболее употребительные способы передачи параметров – по значению (функция получает копию аргумента) и по ссылке (функция получает указатель на аргумент). В Python функция получает копии аргументов,

но аргументы всегда являются ссылками. Поэтому значение объекта, на который указывает ссылка, может измениться, если объект изменяемый, но его идентификатор – никогда. Кроме того, поскольку функция получает копию ссылки, переданной в аргументе, перепривязка не видна за пределами функции. Я позаимствовал термин *вызов по соиспользованию*, прочитав материал на эту тему в книге Michael L. Scott «Programming Language Pragmatics», издание 3 (Morgan Kaufmann), особенно раздел 8.3.1 «Способы передачи параметров».

Часть II

Функции как объекты

Глава 7

Функции как полноправные объекты

> Я никогда не считал, что на Python оказали заметное влияние функциональные языки, что бы кто об этом ни говорил или ни думал. Я был значительно лучше знаком с императивными языками типа C и Algol 68 и, хотя сделал функции полноправными объектами, никогда не рассматривал Python как язык функционального программирования.
>
> – Гвидо ван Россум, пожизненный великодушный диктатор Python[1]

Функции в Python – полноправные объекты. Теоретики языков программирования определяют «полноправный объект» как элемент программы, обладающий следующими свойствами:

- может быть создан во время выполнения;
- может быть присвоен переменной или полю структуры данных;
- может быть передан функции в качестве аргумента;
- может быть возвращен функцией в качестве результата.

Целые числа, строки и словари – все это тоже примеры полноправных объектов в Python, так что ничего необычного тут нет. Реализация функций как полноправных объектов – необходимое свойство функциональных языков, например Clojure, Elixir и Haskell. Однако полноправные функции настолько полезны, что вошли и в другие популярные языки, в т. ч. JavaScript, Go и Java (начиная с JDK 8), ни один из которых не называет себя «функциональным».

В этой главе, да и в части III в целом мы будем изучать практические последствия обращения с функциями как с объектами.

Термин «полноправные функции» широко используется как сокращение фразы «функции как полноправные объекты». Он не совсем точен, потому что наводит на мысль о некоей «элите» среди функций. В Python все функции полноправные.

[1] «Origins of Python's Functional Features» (*http://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html*), из блога Гвидо «История Python».

Что нового в этой главе

Раздел «Девять видов вызываемых объектов» в первом издании назывался «Семь видов вызываемых объектов». Новыми являются платформенные сопрограммы и асинхронные генераторы, добавленные в версиях Python 3.5 и 3.6 соответственно. Те и другие рассматриваются в главе 21, но для полноты картины упомянуты здесь вместе с другими вызываемыми объектами.

«Чисто позиционные параметры» – новый раздел, посвященный средству, добавленному в Python 3.8.

Я перенес обсуждение доступа к аннотациям функций во время выполнения в раздел «Чтение аннотаций типов во время выполнения» главы 15. Когда я работал над первым изданием, документ PEP 484 «Type Hints» (*https://peps.python.org/pep-0484/*) еще находился на этапе рассмотрения, и аннотации использовались разными людьми по-разному. Начиная с версии Python 3.5 аннотации должны соответствовать документу PEP 484. Поэтому лучше всего поместить этот материал туда, где обсуждаются аннотации типов.

В первом издании книги были разделы об интроспекции объектов-функций, которые я счел слишком низкоуровневыми и отвлекающими внимание от основного предмета данной главы. Я объединил все эти разделы в статью «Introspection of Function Parameters» и поместил ее на сайт *fluentpython.com* (*https://www.fluentpython.com/extra/function-introspection/*).

Теперь поговорим о том, почему функции в Python являются полноправными объектами.

Обращение с функцией как с объектом

В сеансе оболочки в примере 7.1 показано, что функции в Python – объекты. Мы создаем функцию, вызываем ее, читаем ее атрибут `__doc__` и проверяем, что сам объект-функция является экземпляром класса `function`.

Пример 7.1. Создаем и тестируем функцию, затем читаем ее атрибут `__doc__` и опрашиваем тип

```
>>> def factorial(n): ❶
...     """возвращает n!"""
...     return 1 if n < 2 else n * factorial(n - 1)
...
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> factorial.__doc__ ❷
'returns n!'
>>> type(factorial) ❸
<class 'function'>
```

❶ Это сеанс оболочки, т. е. мы создаем функцию «во время выполнения».
❷ `__doc__` – один из атрибутов объектов-функций.
❸ `factorial` – экземпляр класса `function`.

Атрибут `__doc__` служит для генерации текста справки по объекту. В интерактивной оболочке Python команда `help(factorial)` выводит информацию, показанную на рис. 7.1.

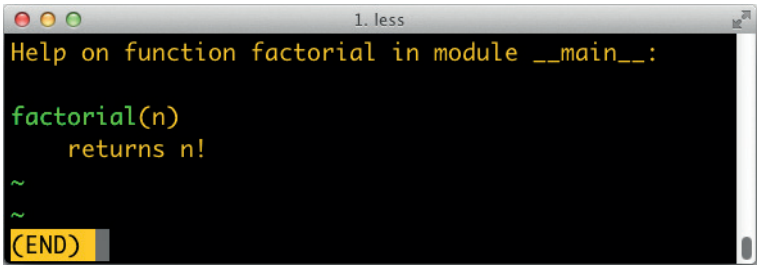


Рис. 7.1. Справка по функции `factorial`. Текст берется из атрибута `__doc__` объекта-функции

Из примера 7.2 видна «полноправность» объекта-функции. Мы можем присвоить функцию переменной `fact` и вызвать ее по имени. Можем передать функцию `factorial` в виде аргумента функции `map`. Функция `map` возвращает итерируемый объект, каждый элемент которого – результат применения первого аргумента (функции) к последовательным элементам второго аргумента (итерируемого объекта), в данном случае `range(10)`.

Пример 7.2. Использование функции под другим именем и передача функции в качестве аргумента

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
>>> map(factorial, range(11))
<map object at 0x...>
>>> list(map(factorial, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Полноправность функций открывает возможности для программирования в функциональном стиле. Один из отличительных признаков функционального программирования (*https://en.wikipedia.org/wiki/Functional_programming*) – использование функций высшего порядка.

Функции высшего порядка

Функцией высшего порядка называется функция, которая принимает функцию в качестве аргумента или возвращает в качестве значения. Примером может служить функция `map` из примера 7.2. Другой пример – встроенная функция `sorted`: ее необязательный аргумент `key` позволяет задать функцию, которая применяется к каждому сортируемому элементу, как было показано в разделе «Метод list.sort и встроенная функция sorted» главы 2. Например, чтобы отсортировать список слов по длине, достаточно передать функцию `len` в качестве аргумента `key`, как в примере 7.3.

Пример 7.3. Сортировка списка слов по длине

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry',
'banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

В роли ключа может выступать любая функция с одним аргументом. Например, для создания словаря рифм полезно отсортировать слова в обратном порядке букв. Обратите внимание, что в примере 7.4 сами слова не изменяются, но поскольку они отсортированы в обратном порядке букв, то все ягоды (berry) оказались рядом.

Пример 7.4. Сортировка списка слов в обратном порядке букв

```
>>> def reverse(word):
...     return word[::-1]
>>> reverse('testing')
'gnitset'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

В функциональной парадигме программирования хорошо известны следующие функции высшего порядка: `map`, `filter`, `reduce`, `apply`. Функция `apply` была объявлена нерекомендуемой в версии Python 2.3 и исключена из Python 3, потому что в ней отпала необходимость. Чтобы вызвать функцию с динамическим набором аргументов, достаточно написать `fn(*args, **keywords)` вместо `apply(fn, args, kwargs)`.

Функции `map`, `filter` и `reduce` пока никуда не делись, но, как показано в следующем разделе, в большинстве случаев им есть лучшие альтернативы.

Современные альтернативы функциям map, filter и reduce

В функциональных языках программирования обычно имеются функции высшего порядка `map`, `filter` и `reduce` (иногда под другими именами). Функции `map` и `filter` по-прежнему встроены в Python 3, но с появлением списковых включений и генераторных выражений потеряли былую значимость. Как списковое включение, так и генераторное выражение могут сделать то же, что комбинация `map` и `filter`, только код будет выглядеть понятнее. Взгляните на пример 7.5.

Пример 7.5. Списки факториалов, порожденные функциями `map` и `filter`, а также альтернатива в виде спискового включения

```
>>> list(map(factorial, range(6))) ❶
[1, 1, 2, 6, 24, 120]
>>> [factorial(n) for n in range(6)] ❷
[1, 1, 2, 6, 24, 120]
>>> list(map(factorial, filter(lambda n: n % 2, range(6)))) ❸
[1, 6, 120]
>>> [factorial(n) for n in range(6) if n % 2] ❹
[1, 6, 120]
>>>
```

❶ Построить список факториалов от 0! до 5!.
❷ Та же операция с помощью спискового включения.
❸ Список факториалов нечетных чисел до 5!, построенный с использованием `map` и `filter`.
❹ Списковое включение делает то же самое, заменяя `map` и `filter` и делая ненужным лямбда-выражение.

В Python 3 функции `map` и `filter` возвращают генераторы – вариант итератора – поэтому безо всяких проблем могут быть заменены генераторным выражением (в Python 2 эти функции возвращали списки, поэтому их ближайшим аналогом было списковое включение).

Функция `reduce`, которая в Python 3 была встроенной, теперь «понижена в звании» и перенесена в модуль `functools`. В той ситуации, где она чаще всего применялась, а именно для суммирования, удобнее встроенная функция `sum`, включенная в версию Python 2.3 в 2003 году. Она дает большой выигрыш в плане удобочитаемости и производительности (см. пример 7.6).

Пример 7.6. Суммирование целых чисел до 99 с помощью `reduce` и `sum`

```
>>> from functools import reduce ❶
>>> from operator import add ❷
>>> reduce(add, range(100)) ❸
4950
>>> sum(range(100)) ❹
4950
>>>
```

❶ Начиная с версии Python 3.0 функция `reduce` больше не является встроенной.
❷ Импортировать модуль `add`, чтобы не создавать функцию для сложения двух чисел.
❸ Вычислить сумму целых чисел, не больших 99.
❹ Решение той же задачи с помощью функции `sum`; импортировать `reduce` и `add` больше не нужно.

Общая идея функций `sum` и `reduce` – применить некую операцию к каждому элементу последовательности с аккумулированием результатов и тем самым свести (редуцировать) последовательность значений к одному.

Редуцирующими являются также встроенные функции `all` и `any`:

`all(iterable)`

Возвращает `True`, если каждый элемент объекта `iterable` «похож на истинный»; `all([])` возвращает `True`.

`any(iterable)`

Возвращает `True`, если хотя бы один элемент объекта `iterable` «похож на истинный»; `all([])` возвращает `False`.

Более полное объяснение `reduce` я приведу в разделе «Vector, попытка № 4: хеширование и ускорение оператора ==» главы 12, где будет подходящий кон-

текст для использования этой функции. А в разделе «Функции редуцирования итерируемого объекта» главы 17, где основной темой обсуждения будут итерируемые объекты, мы подведем итоги.

Иногда для передачи функциям высшего порядка удобно создать небольшую одноразовую функцию. Для этого и предназначены анонимные функции.

Анонимные функции

Ключевое слово `lambda` служит для создания анонимной функции внутри выражения Python.

Однако в силу простоты синтаксиса тело лямбда-функции может быть только чистым выражением. Иными словами, тело `lambda` не может содержать других предложений Python, таких как `while`, `try` и т. д. Присваивание оператором `=` – тоже предложение, поэтому внутри `lambda` его быть не может. Можно использовать новый синтаксис выражения присваивания `:=`, но если вы испытываете в этом необходимость, то, наверное, ваше лямбда-выражение слишком сложное и читать его будет трудно, поэтому лучше преобразовать его в обычную функцию, начинающуюся с `def`.

Особенно удобны анонимные функции в списке аргументов функции высшего порядка. Так, в примере 7.7 код построения словаря рифм из примера 7.4 переписан с помощью `lambda`, без определения функции `reverse`.

Пример 7.7. Сортировка списка слов в обратном порядке букв с помощью `lambda`

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry',
'banana']
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

Помимо задания аргументов функций высшего порядка, анонимные функции редко используются в Python. Из-за синтаксических ограничений нетривиальные лямбда-выражения либо не работают, либо оказываются малопонятны. Если `lambda` читается с трудом, то настоятельно рекомендую последовать совету Фредрика Лундха.

Рецепт рефакторинга лямбда-выражений Лундха

Если вы не можете понять какой-то фрагмент кода из-за использования в нем `lambda`, последуйте совету Фредрика Лундха:

1. Напишите комментарий, объясняющий, что делает `lambda`.
2. Внимательно изучите этот комментарий и придумайте имя, в котором заключалась бы суть изложенного в нем.
3. Преобразуйте `lambda` в предложение `def`, указав придуманное вами имя.
4. Удалите комментарий.

Эти шаги взяты из документа «Functional Programming HOWTO» (*http://docs.python.org/3/howto/functional.html*), который должен прочитать каждый.

Конструкция `lambda` – не более чем синтаксический сахар: лямбда-выражение создает объект-функцию точно так же, как предложение `def`. Это лишь один из нескольких видов вызываемых объектов в Python. В следующем разделе рассмотрены все.

Девять видов вызываемых объектов

Оператор вызова `()` можно применять не только к функциям. Чтобы понять, является ли объект вызываемым, воспользуйтесь встроенной функцией `callable()`. В документации по модели данных Python для версии 3.9 перечислено девять вызываемых типов.

Пользовательские функции

Создаются с помощью предложения `def` или лямбда-выражений.

Встроенные функции

Функции, написанные на языке C (в случае CPython), например `len` или `time.strftime`.

Встроенные методы

Методы, написанные на C, например `dict.get`.

Методы

Функции, определенные в теле класса.

Классы

При вызове класс выполняет свой метод `__new__`, чтобы создать экземпляр, затем вызывает метод `__init__` для его инициализации и, наконец, возвращает экземпляр вызывающей программе. Поскольку в Python нет оператора `new`, вызов класса аналогичен вызову функции[1].

Экземпляры классов

Если в классе определен метод `__call__`, то его экземпляры можно вызывать как функции – это тема следующего раздела.

Генераторные функции

Функции или методы, в теле которых используется ключевое слово `yield`. При вызове генераторная функция возвращает объект-генератор.

Платформенные сопрограммы

Функции или методы, определенные с помощью `async def`. При вызове они возвращают объект сопрограммы. Добавлены в Python 3.5.

Асинхронные генераторные функции

Функции или методы, определенные с помощью `async def`, в теле которых используется ключевое слово `yield`. При вызове генераторная функция возвращает асинхронный генератор с `async for`. Добавлены в Python 3.6.

[1] Обычно при вызове класса создается экземпляр именно этого класса, но такое поведение можно изменить, переопределив метод `__new__`. Соответствующий пример будет приведен в разделе «Гибкое создание объектов с помощью метода `__new__`» главы 22.

Генераторы, платформенные сопрограммы и асинхронные генераторные функции отличаются от других вызываемых объектов тем, что возвращаемые ими значения являются не данными приложения, а объектами, которые нуждаются в последующей обработке, в процессе которой отдают данные приложению или выполняют полезную работу. Генераторные функции возвращают итераторы. Им посвящена глава 17. Платформенные сопрограммы и асинхронные генераторные функции возвращают объекты, которые работают только в сочетании с каким-нибудь каркасом асинхронного программирования, например *asyncio*. Они рассматриваются в главе 21.

Учитывая разнообразие вызываемых типов в Python, самый безопасный способ узнать, является ли объект вызываемым, – воспользоваться встроенной функцией `callable()`:

```
>>> abs, str, 'Ni!'
(<built-in function abs>, <class 'str'>, 'Ni!')
>>> [callable(obj) for obj in (abs, str, 'Ni!')]
[True, True, False]
```

Теперь займемся созданием экземпляров классов, ведущих себя как вызываемые объекты.

Пользовательские вызываемые типы

Мало того что в Python функции являются настоящими объектами, так еще и любой объект можно заставить вести себя как функция. Для этого нужно лишь реализовать метод экземпляра `__call__`.

В примере 7.8 реализован класс `BingoCage`. Экземпляр этого класса строится из любого итерируемого объекта и хранит внутри себя список элементов в случайном порядке. При вызове экземпляра из списка удаляется один элемент[1].

Пример 7.8. bingocall.py: экземпляр `BingoCage` делает всего одну вещь: выбирает элементы из перемешанного списка

```
import random

class BingoCage:

    def __init__(self, items):
        self._items = list(items) ❶
        random.shuffle(self._items) ❷

    def pick(self): ❸
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage') ❹

    def __call__(self): ❺
        return self.pick()
```

[1] Зачем создавать класс `BingoCage`, если уже есть функция `random.choice`? Дело в том, что функция `choice` может вернуть один и тот же элемент несколько раз, поскольку выбранный элемент не удаляется из коллекции. Вызов `BingoCage` никогда не возвращает дубликатов, если, конечно, все значения в коллекции уникальные.

❶ Метод `__init__` принимает произвольный итерируемый объект; создание локальной копии предотвращает изменение списка, переданного в качестве аргумента.
❷ Метод `shuffle` гарантированно работает, потому что `self._items` – объект типа `list`.
❸ Основной метод.
❹ Возбудить исключение со специальным сообщением, если список `self._items` пуст.
❺ Позволяет писать просто `bingo()` вместо `bingo.pick()`.

Ниже приведена простая демонстрация этого кода. Обратите внимание, что объект `bingo` можно вызывать как функцию, и встроенная функция `callable()` распознает его как вызываемый объект:

```
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
1
>>> bingo()
0
>>> callable(bingo)
True
```

Класс, в котором реализован метод `__call__`, – простой способ создать похожий на функцию объект, обладающий внутренним состоянием, которое должно сохраняться между вызовами, как, например, остающиеся элементы в `BingoCage`. Еще один хороший пример – декоратор. Декоратор должен быть вызываемым объектом, и иногда удобно иметь возможность «запоминать» что-то между вызовами декоратора (например, в случае кеширования результатов длительных вычислений для последующего использования) или разбить сложную реализацию на несколько методов.

Функциональный подход к созданию функций, имеющих внутреннее состояние, дают замыкания. Замыкания, как и декораторы, рассматриваются в главе 9.

Теперь исследуем развитый синтаксис, предлагаемый Python для объявления формальных параметров функции и передачи в них фактических аргументов.

От позиционных к чисто именованным параметрам

Одна из самых замечательных особенностей функций в Python – чрезвычайно гибкий механизм обработки параметров, дополненный в Python 3 чисто именованными аргументами. С этой темой тесно связано использование `*` и `**` для распаковки итерируемых объектов и отображений в отдельные аргументы при вызове функции. Чтобы понять, как это выглядит на практике, взгляните на код в примере 7.9 и результат его выполнения в примере 7.10.

Пример 7.9. Функция `tag` генерирует HTML; чисто именованный аргумент `class_` служит для передачи атрибута «class». Это обходное решение необходимо, потому что в Python `class` – зарезервированное слово

```
def tag(name, *content, class_=None, **attrs):
    """Сгенерировать один или несколько HTML-тегов"""
```

```
    if class_ is not None:
        attrs['class'] = class_
    attr_pairs = (f' {attr}="{value}"' for attr, value
                  in sorted(attrs.items()))
    attr_str = ''.join(attr_pairs)
    if content:
        elements = (f'<{name}{attr_str}>{c}</{name}>'
                    for c in content)
        return '\n'.join(elements)
    else:
        return f'<{name}{attr_str} />'
```

Функцию `tag` можно вызывать различными способами, как показано в примере 7.10.

Пример 7.10. Некоторые из многочисленных способов вызвать функцию `tag` из примера 7.9

```
>>> tag('br') ❶
'<br />'
>>> tag('p', 'hello') ❷
'<p>hello</p>'
>>> print(tag('p', 'hello', 'world'))
<p>hello</p>
<p>world</p>
>>> tag('p', 'hello', id=33) ❸
'<p id="33">hello</p>'
>>> print(tag('p', 'hello', 'world', class_='sidebar')) ❹
<p class="sidebar">hello</p>
<p class="sidebar">world</p>
>>> tag(content='testing', name="img") ❺
'<img content="testing" />'
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
... 'src': 'sunset.jpg', 'class': 'framed'}
>>> tag(**my_tag) ❻
'<img class="framed" src="sunset.jpg" title="Sunset Boulevard"/>
```

❶ При задании одного позиционного аргумента порождает пустой тег с таким именем.

❷ Любое число аргументов после первого поглощается конструкцией `*content` и помещается в кортеж.

❸ Именованные аргументы, которые не перечислены явно в сигнатуре функции `tag`, поглощаются конструкцией `**attrs` и помещаются в словарь.

❹ Параметр `class` можно передать только с помощью именованного аргумента.

❺ Первый позиционный аргумент тоже можно передать как именованный.

❻ Если словарю `my_tag` предшествуют две звездочки `**`, то все его элементы передаются как отдельные аргументы, затем некоторые связываются с именованными параметрами, а остальные поглощаются конструкцией `**attrs`. В этом случае в словаре аргументов может присутствовать ключ `'class'`, потому что это строка, так что конфликта с зарезервированным словом `class` нет.

Чисто именованные аргументы – возможность, появившаяся в Python 3. В примере 7.9 параметр `class_` может быть передан только как именованный

аргумент – он никогда не поглощается неименованными позиционными аргументами. Чтобы задать чисто именованные аргументы в определении функции, указывайте их после аргумента с префиксом `*`. Если вы вообще не хотите поддерживать позиционные аргументы, оставив тем не менее возможность, задавать чисто именованные, включите в сигнатуру звездочку `*` саму по себе:

```
>>> def f(a, *, b):
...     return a, b
...
>>> f(1, b=2)
(1, 2)
>>> f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 1 positional argument but 2 were given
```

Отметим, что у чисто именованных аргументов может и не быть значения по умолчанию, они могут быть обязательными, как `b` в предыдущем примере.

Чисто позиционные параметры

Начиная с версии Python 3.8 в сигнатурах пользовательских функций можно задавать чисто позиционные параметры. Такая возможность всегда существовала для встроенных функций, например `divmod(a, b)`, которую можно вызывать только с позиционными параметрами, но не в виде `divmod(a=10, b=4)`.

Чтобы определить функцию, требующую чисто позиционных параметров, используйте символ `/` в списке параметров.

Следующий пример из раздела документации «What's New In Python 3.8» (*https://docs.python.org/3/whatsnew/3.8.html#positional-only-parameters*) показывает, как эмулировать встроенную функцию `divmod`:

```
def divmod(a, b, /):
    return (a // b, a % b)
```

Все аргументы слева от `/` чисто позиционные. После `/` можно задавать и другие аргументы, они будут работать, как обычно.

Наличие `/` в списке параметров считается синтаксической ошибкой в версии Python 3.7 и более ранних.

Например, рассмотрим функцию `tag` из примера 7.9. Если мы хотим, чтобы параметр `name` был чисто позиционным, то можем добавить после него `/` в сигнатуре функции.

```
def tag(name, /, *content, class_=None, **attrs):
    ...
```

Другие примеры чисто позиционных параметров можно найти в разделе документации «What's New In Python 3.8» и в документе PEP 570 (*https://peps.python.org/pep-0570/*).

Разобравшись с гибкими средствами объявления аргументов, мы посвятим оставшуюся часть главы рассмотрению наиболее полезных пакетов, включен-

ных в стандартную библиотеку ради поддержки функционального стиля программирования.

Пакеты для функционального программирования

Хотя Гвидо ясно дал понять, что Python не задумывался как язык функционального программирования, в нем тем не менее можно применять функциональный стиль кодирования – благодаря полноправным функциям, сопоставлению с образцом и таким пакетам, как `operator` и `functools`, которые мы рассмотрим в следующих двух разделах.

Модуль operator

В функциональном программировании часто бывает удобно использовать арифметический оператор как функцию. Пусть, например, требуется перемножить последовательность чисел для нерекурсивного вычисления факториала. Для суммирования можно воспользоваться функцией `sum`, но аналогичной функции для умножения не существует. Можно было бы применить функцию `reduce`, как было показано в разделе «Современные альтернативы функциям map, filter и reduce» выше, но для этого необходима функция умножения двух элементов последовательности. В примере 7.11 показано, как решить эту задачу с помощью `lambda`.

Пример 7.11. Вычисление факториала с помощью `reduce` и анонимной функции

```
from functools import reduce

def factorial(n):
    return reduce(lambda a, b: a*b, range(1, n+1))
```

Чтобы избавить нас от необходимости писать тривиальные анонимные функции вида `lambda a, b: a*b`, модуль `operator` предоставляет функции, эквивалентные многим арифметическим операторам. С его помощью пример 7.11 можно переписать следующим образом.

Пример 7.12. Вычисление факториала с помощью `reduce` и `operator.mul`

```
from functools import reduce
from operator import mul

def factorial(n):
    return reduce(mul, range(1, n+1))
```

Модуль `operator` включает также функции для выборки элементов из последовательностей и чтения атрибутов объектов: `itemgetter` и `attrgetter` строят специализированные функции для выполнения этих действий.

В примере 7.13 показано типичное применение `itemgetter`: сортировка списка кортежей по значению одного поля. В этом примере печатаются города, отсортированные по коду страны (поле 1). По существу, `itemgetter(1)` создает функцию, которая получает коллекцию и возвращает элемент с индексом 1. Это проще писать и читать, чем выражение `lambda fields: fields[1]`, которое делает то же самое.

Пример 7.13. Результат применения `itemgetter` для сортировки списка кортежей (данные взяты из примера 2.8)

```
>>> metro_data = [
... ('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)),
... ]
>>>
>>> from operator import itemgetter
>>> for city in sorted(metro_data, key=itemgetter(1)):
...     print(city)
...
('São Paulo', 'BR', 19.649, (-23.547778, -46.635833))
('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
('New York-Newark', 'US', 20.104, (40.808611, -74.020386))
```

Если передать функции `itemgetter` несколько индексов, то она построит функцию, которая возвращает кортеж, содержащий выбранные значения. Это полезно, когда нужно сортировать по нескольким ключам.

```
>>> for city in metro_data:
...     print(cc_name(city))
...
('JP', 'Tokyo')
('IN', 'Delhi NCR')
('MX', 'Mexico City')
('US', 'New York-Newark')
('BR', 'São Paulo')
>>>
```

Поскольку `itemgetter` пользуется оператором `[]`, то поддерживает не только последовательности, но и отображения, да и вообще любой класс, в котором реализован метод `__getitem__`.

Близким родственником `itemgetter` является функция `attrgetter`, которая создает функции для извлечения атрибутов объекта по имени. Если передать `attrgetter` несколько имен атрибутов, то она также создаст функцию, возвращающую кортеж значений. Кроме того, если имя аргумента содержит точки, то `attrgetter` обойдет вложенные объекты для извлечения атрибута. Описанные возможности продемонстрированы в примере 7.14. Сеанс оболочки получился довольно длинным, потому что нам пришлось построить вложенную структуру для демонстрации обработки имен атрибутов с точкой.

Пример 7.14. Применение `attrgetter` для обработки ранее определенного списка именованных кортежей `metro_data` (тот же список, что в примере 7.13)

```
>>> LatLon = namedtuple('LatLon', 'lat lon') ❶
>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord') ❷
>>> metro_areas = [Metropolis(name, cc, pop, LatLon(lat, lon)) ❸
...     for name, cc, pop, (lat, lon) in metro_data]
>>> metro_areas[0]
Metropolis(name='Tokyo', cc='JP', pop=36.933,
```

```
coord=LatLon(lat=35.689722,
lon=139.691667))
>>> metro_areas[0].coord.lat ❹
35.689722
>>> from operator import attrgetter
>>> name_lat = attrgetter('name', 'coord.lat') ❺
>>>
>>> for city in sorted(metro_areas, key=attrgetter('coord.lat')): ❻
...     print(name_lat(city)) ❼
...
('São Paulo', -23.547778)
('Mexico City', 19.433333)
('Delhi NCR', 28.613889)
('Tokyo', 35.689722)
('New York-Newark', 40.808611)
```

❶ Определить именованный кортеж `LatLon`.
❷ Определить также `Metropolis`.
❸ Построить список `metro_areas`, содержащий экземпляры `Metropolis`; обратите внимание на распаковку именованного кортежа для извлечения `(lat, lon)` и использование этих данных для построения объекта `LatLon`, являющегося значением атрибута `coord` объекта `Metropolis`.
❹ Получить широту из элемента `metro_areas[0]`.
❺ Определить `attrgetter` для выборки атрибута `name` и вложенного атрибута `coord.lat`.
❻ Снова использовать `attrgetter` для сортировки списка городов по широте.
❼ Использовать определенный выше `attrgetter` для показа только названия и широты города.

Ниже приведен неполный список функций в модуле `operator` (имена, начинающиеся знаком подчеркивания, опущены, потому что такие функции содержат детали реализации):

```
>>> [name for name in dir(operator) if not name.startswith('_')]
['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imatmul',
'imod', 'imul', 'index', 'indexOf', 'inv', 'invert', 'ior',
'ipow', 'irshift', 'is_', 'is_not', 'isub', 'itemgetter',
'itruediv', 'ixor', 'le', 'length_hint', 'lshift', 'lt',
'matmul',
'methodcaller', 'mod', 'mul', 'ne', 'neg', 'not_', 'or_', 'pos',
'pow', 'rshift', 'setitem', 'sub', 'truediv', 'truth', 'xor']
```

Назначение большинства из этих 54 функций очевидно. Функции, имена которых начинаются с `i` и далее содержат имя оператора, например `iadd`, `iand` и т. д., соответствуют составным операторам присваивания: `+=`, `&=` и т. д. Они изменяют свой первый аргумент на месте, если это изменяемый объект; в противном случае функция работает так же, как аналогичная без префикса `i`: просто возвращает результат операции.

Из прочих функций мы рассмотрим только `methodcaller`. Она похожа на `attrgetter` и `itemgetter` в том смысле, что на лету создает функцию. Эта функция вызывает метод по имени для объекта, переданного в качестве аргумента (пример 7.15).

Пример 7.15. Демонстрация `methodcaller`: во втором тесте показано связывание дополнительных аргументов

```
>>> from operator import methodcaller
>>> s = 'The time has come'
>>> upcase = methodcaller('upper')
>>> upcase(s)
'THE TIME HAS COME'
>>> hyphenate = methodcaller('replace', ' ', '-')
>>> hyphenate(s)
'The-time-has-come'
```

Первый тест в примере 7.15 просто показывает, как работает функция `methodcaller`, но вообще-то если нужно использовать метод `str.upper` как функцию, то можно просто вызвать его от имени класса `str`, передав строку в качестве аргумента:

```
>>> str.upper(s)
'THE TIME HAS COME'
```

Второй тест показывает, что `methodcaller` позволяет также фиксировать некоторые аргументы – так же, как функция `functools.partial`. Это и будет нашей следующей темой.

Фиксация аргументов с помощью functools.partial

В модуле `functools` собраны некоторые функции высшего порядка. Мы видели функцию `reduce` в разделе «Современные альтернативы функциям map, filter и reduce» выше. Еще одна функция – `partial`: получив на входе вызываемый объект, она порождает новый вызываемый объект, в котором некоторые аргументы исходного объекта связаны с заранее определенными значениями. Это полезно для адаптации функции, принимающей один или несколько аргументов, к API, требующему обратного вызова функции с меньшим числом аргументов. Тривиальная демонстрация приведена в примере 7.16.

Пример 7.16. Использование `partial` позволяет вызывать функцию с двумя аргументами там, где требуется вызываемый объект с одним аргументом

```
>>> from operator import mul
>>> from functools import partial
>>> triple = partial(mul, 3) ❶
>>> triple(7) ❷
21
>>> list(map(triple, range(1, 10))) ❸
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

❶ Создать новую функцию `triple` из `mul`, связав первый аргумент со значением `3`.
❷ Протестировать ее.
❸ Использовать `triple` совместно с `map`; `mul` в этом примере не смогла бы работать с `map`.

Более полезный пример относится к функции `unicode.normalize`, с которой мы встречались в разделе «Нормализация Unicode для правильного сравнения» главы 4. Для многих языков перед сравнением или сохранением строки рекомендуется нормализовывать с помощью вызова `unicode.normalize('NFC', s)`. Если

это приходится делать часто, то удобно завести функцию `nfc`, как показано в примере 7.17.

Пример 7.17. Построение вспомогательной функции нормализации Unicode-строк с помощью `partial`

```
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> s1 == s2
False
>>> nfc(s1) == nfc(s2)
True
```

Функция `partial` принимает в первом аргументе вызываемый объект, а за ним – произвольное число позиционных и именованных аргументов, подлежащих связыванию.

В примере 7.18 демонстрируется использование `partial` совместно с функцией `tag` из примера 7.9 для фиксации одного позиционного и одного именованного аргумента.

Пример 7.18. Применение `partial` к функции `tag` из примера 7.9

```
>>> from tagger import tag
>>> tag
<function tag at 0x10206d1e0> ❶
>>> from functools import partial
>>> picture = partial(tag, 'img', class_='pic-frame') ❷
>>> picture(src='wumpus.jpeg')
'<img class="pic-frame" src="wumpus.jpeg" />' ❸
>>> picture
functools.partial(<function tag at 0x10206d1e0>, 'img', class_='pic-frame') ❹
>>> picture.func ❺
<function tag at 0x10206d1e0>
>>> picture.args
('img',)
>>> picture.keywords
{'cls': 'pic-frame'}
```

❶ Импортировать функцию `tag` из примера 7.9 и показать ее идентификатор.
❷ Создать функцию `picture` из `tag`, зафиксировав значение первого позиционного аргумента – `'img'` и значение именованного параметра `class_` – `'pic-frame'`.
❸ Функция `picture` работает, как и ожидалось.
❹ `partial()` возвращает объект `functools.partial`[1].
❺ У объекта `functools.partial` есть атрибуты, дающие доступ к исходной функции и фиксированным аргументам.

[1] Из исходного кода (*https://github.com/python/cpython/blob/0bbf30e2b910bc9c5899134ae9d73a8df968 da35/Lib/functools.py#L341*) в скрипте `functools.py` становится ясно, что класс `functools.partial` реализован на C и используется по умолчанию. Если он недоступен, то начиная с версии Python 3.4 в модуле functools имеется реализация `partial` на чистом Python.

Функция `functools.partialmethod` делает то же, что `partial`, но предназначена для работы с методами.

Модуль `functools` включает также функции высшего порядка, рассчитанные на использование в качестве декораторов, в частности `cache` и `singledispatch`. Мы рассмотрим их в главе 9, где также обсуждается, как реализовать пользовательские декораторы.

Резюме

Целью этой главы было исследование функций как полноправных объектов Python. Идея в том, что функции можно присваивать переменным, передавать другим функциям, сохранять в структурах данных, а также получать атрибуты функций, что позволяет каркасам и инструментальным средствам принимать те или иные решения.

Функции высшего порядка, основа функционального программирования, часто используются в программах на Python. Встроенные функции `sorted`, `min`, `max` и `functools.partial` – примеры распространенных функций высшего порядка. Функции `map`, `filter` и `reduce` употребляются реже, чем в былые времена, – благодаря списковому включению (и аналогичным конструкциям, например генераторным выражениям) и наличию встроенных редуцирующих функций типа `sum`, `all` и `any`.

Начиная с версии Python 3.6 в языке имеется девять видов вызываемых объектов: от простых функций, создаваемых с помощью `lambda`, до экземпляров классов, в которых реализован метод `__call__`. Генераторы и сопрограммы также являются вызываемыми объектами, хотя и совсем не похожими на другие. Все вызываемые объекты распознаются встроенной функцией `callable()`. Любой вызываемый объект поддерживает общий развитый синтаксис объявления формальных параметров, в том числе чисто именованные параметры, чисто позиционные параметры и аннотации.

Наконец, мы рассмотрели несколько функций из модуля `operator` и функцию `functools.partial`, которая упрощает функциональное программирование за счет уменьшения потребности в синтаксисе лямбда-выражений.

Дополнительная литература

В следующих двух главах мы продолжим изучение программирования с помощью объектов-функций. Глава 8 посвящена аннотациям типов в параметрах и возвращаемых значениях функций. Тема главы 9 – декораторы, специальный вид функций высшего порядка, и механизм замыкания, благодаря которому декораторы и работают. В главе 10 показано, как полноправные функции упрощают некоторые классические паттерны объектно-ориентированного программирования.

В разделе 3.2 «Иерархия стандартных типов» справочного руководства по языку Python (*https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy*) описаны все девять вызываемых типов, а также остальные встроенные типы.

Глава 7 книги David Beazley, Brian K. Jones «*Python Cookbook*», 3-е издание (O'Reilly), отлично дополняет эту и девятую главу, поскольку к объяснению тех же концепций подходит по-другому.

Если вас интересуют причины, по которым в язык были включены чисто именованные аргументы, а также примеры их использования, обратитесь к документу PEP 3102 «Keyword-Only Arguments» (*https://www.python.org/dev/peps/pep-3102/*).

Отличное введение в функциональное программирование на Python – составленный А. М. Кухлингом документ «Python Functional Programming HOWTO» (*http://docs.python.org/3/howto/functional.html*). Но в основном он посвящен использованию итераторов и генераторов, которые рассматриваются в главе 17.

На заданный на сайте StackOverflow вопрос «Зачем нужна функция functools.partial в Python» (*https://stackoverflow.com/questions/3252228/python-why-is-functools-partial-necessary*) в высшей степени информативный ответ дал Алекс Мартелли, соавтор классической книги «Python in a Nutshell» (O'Reilly).

Размышляя над вопросом, является ли Python функциональным языком, я создал одну из своих любимых презентаций, «Beyond Paradigms», которую представлял на конференциях PyCaribbean, PyBay и PyConDE. См. слайды (*https://speakerdeck.com/ramalho/beyond-paradigms-berlin-edition*) и видео (*https://www.youtube.com/watch?v=bF3a2VYXxa0*) с презентации в Берлине, где я повстречался с Мирославом Седивым и Юргеном Гмахом, которые вошли в число технических рецензентов этой книги.

Поговорим

Является ли Python функциональным языком?

Где-то в 2000 году я проходил обучение на курсах компании Zope Corporation в США, когда в аудиторию заглянул Гвидо ван Россум (он не преподавал). В последовавшей серии вопросов и ответов кто-то спросил, какие функции Python заимствовал из других языков. Гвидо ответил: «Все, что есть хорошего в Python, украдено из других языков».

Шрирам Кришнамурти (Shriram Krishnamurthi), профессор информатики в Брауновском университете, начинает свою статью «Teaching Programming Languages in a Post-Linnaean Age» (*http://cs.brown.edu/~sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/*) такими словами:

> «Парадигмы» языков программирования – отжившее и никому не нужное наследие ушедшего века. Проектировщики современных языков не обращают на них никакого внимания, так почему же на учебных курсах мы так рабски им привержены?

В той же статье упоминается и Python:

> Что еще сказать о таких языках, как Python, Ruby или Perl? У их проектировщиков не было терпения изучать красоты линнеевских иерархий; они брали те функциональные возможности, которые считали нужными, творя смеси, не поддающиеся никакой классификации.

Кришнамурти предлагает не пытаться классифицировать языки, следуя заранее выбранной таксономии, а рассматривать их как агрегаты функциональных возможностей. Его идеи вдохновили меня на создание презентации «Beyond Paradigms», упомянутой в конце раздела «Дополнительная литература».

И хотя Гвидо не ставил такой цели, включение в Python полноправных функций распахнуло двери функциональному программированию. В сообщении «Origins of Python's *Functional* Features» Гвидо пишет, что именно функции `map`, `filter` и `reduce` стали поводом для реализации в Python лямбда-выражений (*http://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html*). Все эти средства предложил для включения в Python 1.0 Амрит Прем (Amrit Prem) в 1994 году, согласно файлу Misc/HISTORY по адресу *http://hg.python.org/cpython/file/ default/Misc/HISTORY* в дереве исходного кода CPython.

Такие функции, как `lambda`, `map`, `filter` и `reduce`, впервые появились в Lisp, первом функциональном языке программирования. Однако в Lisp не налагаются ограничения на то, что можно делать внутри `lambda`, потому что в Lisp любая конструкция является выражением. В Python принят синтаксис, основанный на предложениях; выражения в нем не могут содержать предложения, а многие языковые конструкции являются предложениями – в том числе блок `try/catch`, которого мне особенно не хватает в лямбда-выражениях. Такова цена, которую приходится платить за в высшей степени удобочитаемый синтаксис[1]. У языка Lisp много сильных сторон, но удобочитаемость не из их числа.

По иронии судьбы, заимствование синтаксиса спискового включения из другого функционального языка, Haskell, заметно сократило потребность в `map` и `filter`, да, кстати, и в `lambda`.

Помимо ограничений синтаксиса анонимных функций, основным препятствием для более широкого принятия идиом функционального программирования в Python является отсутствие устранения хвостовой рекурсии – оптимизации, которая уменьшает потребление памяти функцией, выполняющей рекурсивный вызов в конце своего тела. В другом сообщении, «Tail Recursion Elimination» (*http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html)*, Гвидо приводит несколько причин, по которым такая оптимизация плохо подходит для Python. Это сообщение весьма интересно технической аргументацией, но еще более тем, что первые три – самые важные – причины касаются удобства пользования. Тот факт, что использование, изучение и преподавание Python доставляет массу удовольствия, – не случайность. Гвидо специально стремился к этому.

Итак: Python по своему замыслу не является функциональным языком – что бы под этим ни понимать. Python лишь заимствует кое-какие удачные идеи из функциональных языков.

Проблема анонимных функций

Помимо синтаксических ограничений, связанных со спецификой Python, у анонимных функций есть серьезный недостаток в любом языке: отсутствие имени.

И это лишь наполовину шутка. Трассировку стека проще читать, если у функций есть имена. Анонимные функции удобны, когда нужно срезать угол, программисты любят их писать, но иногда слишком увлекаются – особенно если язык и среда поощряют глубокую вложенность анонимных функций, как, скажем, JavaScript в среде Node.js. Большое количество вложенных анонимных функций усложняет отладку и обработку ошибок.

[1] Еще есть проблема потери отступов при копировании кода в веб-форумы, но это я отвлекся.

Асинхронное программирование в Python более структурировано, быть может, потому что ограничения лямбда-выражений препятствуют злоупотреблениям и заставляют искать более явные подходы. Обещания, будущие и отложенные объекты – концепции, используемые в API асинхронного программирования. Наряду с сопрограммами они открывают выход из так называемого «ада обратных вызовов». Обещаю рассказать подробнее об асинхронном программировании в будущем, но эту тему отложу до главы 21.

Глава 8

Аннотации типов в функциях

> Следует также подчеркнуть, что **Python останется динамически типизированным языком, и у авторов нет ни малейшего желания когда-нибудь делать аннотации типов обязательными, пусть даже в силу соглашения.**
>
> – Гвидо ван Россум, Юкка Лехтосало и Лукаш Ланга, PEP 484 «Type Hints»[1]

Аннотации типов – самое большое изменение в истории Python после унификации типов и классов (*https://www.python.org/download/releases/2.2.3/descrintro/*) в Python 2.2 в 2001 году. Однако аннотации типов устраивают не всех пользователей Python, поэтому они должны быть факультативными.

В документе PEP 484 «Type Hints» (*https://peps.python.org/pep-0484/*) описаны синтаксис и семантика явных объявлений типов в аргументах функций, возвращаемых значениях и переменных. Цель – помочь инструментам разработки искать ошибки в кодовых базах на Python с помощью статического анализа, т. е. не выполняя тестов кода.

Основными выгодоприобретателями являются профессиональные программисты, пользующиеся интегрированными средами разработки (IDE) и системами непрерывной интеграции (CI). Анализ преимуществ и недостатков, доказывающий привлекательность аннотаций типов для этой группы, распространяется не на всех пользователей Python.

Круг пользователей Python гораздо шире. Он включает, в частности, научных работников, трейдеров, журналистов, художников, производителей, аналитиков и студентов многих специальностей. Для большинства из них плата за изучение аннотаций типов, скорее всего, окажется несоразмерно высокой, если только они уже незнакомы с каким-нибудь языком, имеющим статические типы, механизм создания подтипов и обобщенные типы. А преимущества для многих из них окажутся не столь ощутимыми, учитывая характер их работы с Python, а также меньший размер кодовой базы и команды – зачастую «команда» состоит из одного человека. Динамическая типизация Python, подразумеваемая по умолчанию, проще и выразительнее, когда нужно написать код для исследования данных и идей, как то бывает в науке о данных, применении компьютеров для творчества и обучения.

Эта глава посвящена аннотациям типов в сигнатурах функций, а в главе 15 мы изучим аннотации типов в контексте классов, а также другие средства, имеющиеся в модуле `typing`.

---

[1] PEP 484 – Type Hints, «Rationale and Goals»; сохранено выделение оригинала.

В этой главе будут рассмотрены следующие вопросы:

- практическое введение в постепенную типизацию на примере Муру;
- дополнительные возможности утиной и номинальной типизации;
- обзор основных категорий типов, которые могут встречаться в аннотациях (это примерно 60 % главы);
- аннотации типов с переменным числом параметров (`*args`, `**kwargs`);
- ограничения и недостатки аннотаций типов и статической типизации.

Что нового в этой главе

Вся эта глава новая. Аннотации типов появились в версии Python 3.5 уже после того, как я закончил работу над первым изданием книги.

С учетом ограничений статической системы типов лучшее, что мог предложить документ PEP 484, – внедрение *системы постепенной типизации*. Начнем с определения того, что этот термин означает.

О постепенной типизации

В документе PEP 484 описано включение *системы постепенных типов* в Python. Из других языков с системой постепенных типов назовем TypeScript от Microsoft, Dart (язык Flutter SDK, созданный Google) и Hack (диалект PHP, поддерживаемый виртуальной машиной HHVM от Facebook). Сама программа проверки типов Муру начиналась как язык: постепенно типизированный диалект Python со своим собственным интерпретатором. Гвидо ван Россум убедил создателя Муру, Юкку Лехтосало, превратить его в инструмент для проверки аннотированного Python-кода.

Система постепенной типизации обладает следующими характеристиками.

Является факультативной

По умолчанию средство проверки типов не должно выдавать предупреждений для кода без аннотаций типов. Вместо этого оно должно предполагать тип `Any`, если не может определить тип объекта. Тип `Any` считается совместимым с любым другим.

Не обнаруживает ошибки типизации во время выполнения

Аннотации типов используются средствами проверки типов, линтерами и IDE для выдачи предупреждений. Они не препятствуют ни передаче несогласованных значений функциям, ни присваиванию их переменным во время выполнения.

Не увеличивает производительность

Аннотации типов предоставляют сведения, которые теоретически могли бы открыть возможность оптимизаций сгенерированного байт-кода, но по состоянию на июль 2021 года такие оптимизации не реализованы ни в одной из известных мне сред выполнения Python[1].

[1] JIT-компилятор, например имеющийся в PyPy, располагает куда более полными данными, чем аннотации типов: он наблюдает за Python-программой во время ее выполнения, видит, какие конкретно типы она использует, и генерирует оптимизированный машинный код именно для этих типов.

Самая важная для пользователей особенность постепенной типизации – тот факт, что аннотации всегда факультативны.

При использовании статической системы типов большинство ограничений на типы выразить легко, многие неудобно, некоторые трудно, а совсем немногие невозможно[1]. Мы без труда сможем написать прекрасный код на Python, с хорошим тестовым покрытием, но при этом не сумеем добавить аннотации типов, которые удовлетворили бы средство проверки. Ничего страшного – просто опустите проблематичные аннотации типов и пускайте в эксплуатацию!

Аннотации типов факультативны на всех уровнях: можно встретить целые пакеты без единой аннотации, можно подавить проверку типов при импорте такого пакета в модуль, где аннотации используются, и можно добавить специальные комментарии, заставляющие средство проверки типов игнорировать конкретные строки кода.

Стремление к стопроцентному покрытию аннотациями типов означает, что вы применяете аннотирование без ясной цели, просто ради красивых показателей. К тому же это будет мешать команде добиться максимума от мощи и гибкости Python. Код без аннотаций типов должен без колебаний приниматься, если аннотации сделали бы API менее удобным для пользователя или без нужды усложнили бы реализацию.

Постепенная типизация на практике

Посмотрим, как постепенная типизация работает на практике. Начнем с простой функции и будем постепенно добавлять в нее аннотации типов, под руководством Муру.

Существует несколько программ проверки типов для Python, совместимых с PEP 484, в т. ч. Google pytype (*https://github.com/google/pytype*), Microsoft Pyright (*https://github.com/Microsoft/pyright), Facebook Pyre (https://pyre-check.org/*) – и это помимо средств, встроенных в IDE типа PyCharm. Я выбрал для примеров Муру (*https://mypy.readthedocs.io/en/stable/*), потому что она самая известная. Но для каких-то проектов или команд другая программа может оказаться более подходящей. К примеру, Pytype проектировалась так, что может работать с кодовыми базами без аннотаций типов и тем не менее давать полезные советы. Она более снисходительна, чем Муру, и умеет сама генерировать аннотации для вашего кода.

Мы будем аннотировать функцию `show_count`, которая возвращает строку, содержащую счетчик и слово в единственном или множественном числе в зависимости от значения счетчика:

```
>>> show_count(99, 'bird')
'99 birds'
```

[1] Например, по состоянию на июль 2021 года рекурсивные типы не поддерживались – см. проблему #182, зарегистрированную для модуля `typing`: «Define a JSON type» (*https://github.com/python/typing/issues/182*), и проблему #731 для Муру «Support recursive types» (*https://github.com/python/mypy/issues/731*).

```
>>> show_count(1, 'bird')
'1 bird'
>>> show_count(0, 'bird')
'no birds'
```

В примере 8.1 приведен исходный код `show_count` без аннотаций.

Пример 8.1. Функция `show_count` из модуля messages.py без аннотаций типов

```
def show_count(count, word):
    if count == 1:
        return f'1 {word}'
    count_str = str(count) if count else 'no'
    return f'{count_str} {word}s'
```

Начинаем работать с Муру

Чтобы приступить к проверке типов, я применил команду `mypy` к модулю *messages.py*:

```
…/no_hints/ $ pip install mypy
[много сообщений опущено...]
…/no_hints/ $ mypy messages.py
Success: no issues found in 1 source file
```

Муру с параметрами по умолчанию не нашла никаких ошибок в примере 8.1.

Я пользуюсь версией Муру 0.910, последней по состоянию на июль 2021 года. Во «Введении» к Муру есть предупреждение, что «официально это бета-версия. Возможны изменения, нарушающие обратную совместимость». Муру выдала как минимум одно сообщение, отличающееся от того, которые я получал в апреле 2020 года, когда писал эту главу. Не исключено, что когда вы будете ее читать, результаты будут отличаться от представленных здесь.

Если в сигнатуре функции нет аннотаций, то Муру по умолчанию игнорирует ее, если не была сконфигурирована иначе.

В примере 8.2 приведены автономные тесты, рассчитанные на `pytest`.

Пример 8.2. Файл messages_test.py без аннотаций типов

```
from pytest import mark

from messages import show_count

@mark.parametrize('qty, expected', [
    (1, '1 part'),
    (2, '2 parts'),
])

def test_show_count(qty, expected):
    got = show_count(qty, 'part')
    assert got == expected

def test_show_count_zero():
    got = show_count(0, 'part')
    assert got == 'no parts'
```

Теперь под чутким руководством Муру добавим аннотации типов.

А теперь построже

Параметр командной строки `--disallow-untyped-defs` заставляет Муру помечать все определения функций, в которых нет аннотаций типов для всех параметров и возвращаемого значения.

Задав `--disallow-untyped-defs` для тестового файла, мы получим три сообщения об ошибках и одно примечание:

```
…/no_hints/ $ mypy --disallow-untyped-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
messages_test.py:10: error: Function is missing a type annotation
messages_test.py:15: error: Function is missing a return type annotation
messages_test.py:15: note: Use "-> None" if function does not return a value
Found 3 errors in 2 files (checked 1 source file)
```

Начиная добавлять постепенную типизацию, я предпочитаю задавать другой параметр: `--disallow-incomplete-defs`. Сначала он не говорит ничего:

```
…/no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
Success: no issues found in 1 source file
```

Теперь попробую аннотировать в `show_count` только возвращаемое значение:

```
def show_count(count, word) -> str:
```

Этого достаточно, чтобы Муру обратила на функцию внимание. Та же команда, что и выше, примененная для проверки *messages_test.py*, побудит Муру снова взглянуть на *messages.py*:

```
…/no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
for one or more arguments
Found 1 error in 1 file (checked 1 source file)
```

Теперь я могу постепенно добавлять аннотации типов к одной функции за другой, не получая предупреждений для еще не аннотированных функций. Вот как выглядит полностью аннотированная сигнатура, удовлетворяющая Муру:

```
def show_count(count: int, word: str) -> str:
```

Вместо того чтобы задавать параметры типа `--disallow-incomplete-defs` в командной строке, можно сохранить те, которыми вы пользуетесь особенно часто, в конфигурационном файле Муру. Параметры могут задаваться глобально или на уровне модуля. Вот как может выглядеть простой файл *mypy.ini* в начале работы:

```
[mypy]
python_version = 3.9
warn_unused_configs = True
disallow_incomplete_defs = True
```

Значение параметра по умолчанию

Функция `show_count` в примере 8.1 работает только с правильными именами существительными. Если множественное число образуется не добавлением

буквы 's', то мы должны дать пользователю возможность самому указать форму множественного числа, например:

```
>>> show_count(3, 'mouse', 'mice')
'3 mice'
```

Давайте займемся «разработкой через типы». Сначала добавим тест, в котором используется этот третий аргумент. Не забудем добавить аннотацию возвращаемого значения в тестовой функции, иначе Муру не станет ее проверять.

```
def test_irregular() -> None:
    got = show_count(2, 'child', 'children')
    assert got == '2 children'
```

Муру обнаруживает ошибку:

```
…/hints_2/ $ mypy messages_test.py
messages_test.py:22: error: Too many arguments for "show_count"
Found 1 error in 1 file (checked 1 source file)
```

Теперь изменим `show_count`, добавив необязательный параметр `plural`, как показано в примере 8.3.

Пример 8.3. Функция `show_count` из модуля hints_2/messages.py с факультативным параметром

```
def show_count(count: int, singular: str, plural: str = '') -> str:
    if count == 1:
        return f'1 {singular}'
    count_str = str(count) if count else 'no'
    if not plural:
        plural = singular + 's'
    return f'{count_str} {plural}'
```

Теперь Муру говорит «Success» – все хорошо.

В коде ниже есть опечатка, которую Python не замечает. Сможете найти ее?

```
def hex2rgb(color=str) -> tuple[int, int, int]:
```

Сообщение об ошибке, выдаваемое Муру, не сильно помогает:

```
colors.py:24: error: Function is missing a type
    annotation for one or more arguments
```

Аннотация типа для аргумента `color` должна иметь вид `color: str`. Я написал `color=str`, а это вообще не аннотация. Такая запись означает, что `color` по умолчанию имеет значение `str`.

По моему опыту, это довольно типичная ошибка, которую легко проглядеть, особенно в сложных аннотациях типов.

Следующие рекомендации считаются признаками хорошего стиля в аннотациях типов:

- между именем параметра и `:` не должно быть пробелов, после `:` должен быть один пробел;
- по обе стороны от знака `=`, предшествующего значению по умолчанию, должно быть по одному пробелу.

С другой стороны, в документе PEP 8 говорится, что вокруг `=` не должно быть пробелов, если для этого конкретного параметра нет аннотации типа.

Стиль кодирования: используйте flake8 и blue

Вместо того чтобы запоминать всякие глупые правила, пользуйтесь инструментами типа *flake8* (*https://pypi.org/project/flake8/*) и *blue* (*https://pypi.org/project/blue/*). *flake8* в числе прочего сообщает о недочетах, связанных со стилем кодирования, а *blue* переписывает исходный код в соответствии с правилами (большинством из них), встроенными в средство форматирования кода *black* (*https://pypi.org/project/black/*).

Если цель – навязать «стандартный» стиль кодирования, то *blue* лучше, чем *black*, потому что следует присущей Python рекомендации использовать по умолчанию одиночные кавычки, а двойные – как альтернативу:

```
>>> "I prefer single quotes"
'I prefer single quotes'
```

Предпочтение одиночным кавычкам встроено в функцию `repr()` и в другие места CPython. Модуль *doctest* (*https://docs.python.org/3/library/doctest.html*) зависит от использования одиночных кавычек в `repr()`.

Одним из авторов *blue* является Барри Уорсоу (*https://wefearchange.org/2020/11/steeringcouncil.rst.html*), соавтор PEP 8, разработчик ядра Python с 1994 года и член Руководящего совета Python с 2019 года по настоящее время (июль 2021 года). Выбирая одиночные кавычки по умолчанию, мы оказываемся в очень хорошей компании.

Если вам позарез необходимо использовать *black*, то задавайте параметр `black -S`. Тогда кавычки в коде не будут изменены.

None в качестве значения по умолчанию

В примере 8.3 параметр `plural` аннотирован типом `str`, а значение по умолчанию равно `''`, так что конфликта типов не возникает.

Мне нравится это решение, но в других контекстах лучше использовать по умолчанию `None`. Если факультативный параметр имеет изменяемый тип, то `None` – единственно разумное значение по умолчанию, как было показано в разделе «Значения по умолчанию изменяемого типа: неудачная мысль» главы 6.

Чтобы значением по умолчанию параметра `plural` было `None`, сигнатура должна выглядеть так:

```
from typing import Optional

def show_count(count: int, singular: str, plural: Optional[str] = None) -> str:
```

Расшифруем:

- `Optional[str]` означает, что `plural` может иметь тип `str` или `None`;
- необходимо явно задать значение по умолчанию `= None`.

Если параметру `plural` не сопоставлено значение по умолчанию, то среда выполнения Python будет рассматривать его как обязательный. Помните: на этапе выполнения аннотации типов игнорируются.

Отметим, что мы должны импортировать `Optional` из модуля `typing`. При импортировании типов рекомендуется использовать синтаксис `from typing import X`, чтобы уменьшить длину сигнатур функций.

`Optional` – не самое лучшее имя, потому что наличие аннотации не делает параметр факультативным. Чтобы параметр стал факультативным, ему нужно назначить значение по умолчанию. `Optional[str]` просто означает, что тип параметра может быть `str` или `NoneType`. В языках Haskell и Elm похожий тип называется `Maybe`.

Теперь, составив первое представление о постепенной типизации, рассмотрим, что концепция *типа* означает на практике.

Типы определяются тем, какие операции они поддерживают

> В литературе есть много определений понятия типа. Здесь мы предполагаем, что тип – это множество значений и набор функций, применимых к этим значениям.
>
> – PEP 483 «The Theory of Type Hints»

На практике полезнее рассматривать набор операций как определяющую характеристику типа[1].

Например, с точки зрения применимых операций, каковы допустимые типы `x` в следующей функции?

```
def double(x):
    return x * 2
```

Параметр `x` может быть числовым (`int`, `complex`, `Fraction`, `numpy.uint32` и т. д.), но может быть также последовательностью (`str`, `tuple`, `list`, `array`), *N*-мерным массивом `numpy.array` или любым другим типом, который реализует или наследует метод `__mul__`, принимающий аргумент типа `int`.

Однако рассмотрим следующую аннотированную функцию `double`. Не обращайте внимания на отсутствие типа возвращаемого значения, давайте сосредоточимся на типе параметра:

```
from collections import abc

def double(x: abc.Sequence):
    return x * 2
```

Средство проверки типа отвергнет этот код. Если вы скажете Муру, что `x` имеет тип `abc.Sequence`, то она пометит выражение `x * 2` как ошибку, потому

[1] В Python нет синтаксиса для управления множеством допустимых значений типа, если не считать типов `Enum`. Например, с помощью аннотаций типов невозможно определить тип `Quantity` как множество целых чисел от 1 до 1000 или тип `AirportCode` как трехбуквенную аббревиатуру. NumPy предлагает типы `uint8`, `int16` и другие машинно-ориентированные числовые типы, но в стандартной библиотеке Python есть только типы с очень небольшим числом значений (`NoneType`, `bool`) или очень большим множеством значений (`float`, `int`, `str`, все кортежи и т. д.).

что абстрактный базовый тип `Sequence` (*https://docs.python.org/3/library/ collections.abc.html#collections-abstract-base-classes*) не реализует и не наследует метод `__mul__`. Во время выполнения этот код будет работать с конкретными последовательностями, такими как `str`, `tuple`, `list`, `array` и т. д., а также с числами, потому на этапе выполнения аннотации типов игнорируются. Но для средства проверки типов важно лишь то, что объявлено явно, а в классе `abc.Sequence` метода `__mul__` нет.

Потому-то этот раздел и называется «Типы определяются тем, какие операции они поддерживают». Среда выполнения Python примет любой объект в качестве аргумента `x` для обоих вариантов функции `double`. Вычисление `x * 2` может закончиться успешно или возбудить исключение `TypeError`, если операция не поддерживается `x`. С другой стороны, Муру, проанализировав аннотированный код `double`, сообщит, что выражение `x * 2` недопустимо, потому что для объявленного типа `x: abc.Sequence` операция не поддерживается.

В системе постепенной типизации сталкиваются два разных взгляда на типы:

Утиная типизация

Этот взгляд принят в Smalltalk – первопроходческом объектно-ориентированном языке, а также в Python, JavaScript и Ruby. У объектов есть типы, но переменные (в т. ч. параметры) не типизированы. На практике не важно, каков объявленный тип объекта, важно лишь, какие операции он фактически поддерживает. Если я могу вызвать метод `birdie.quack()`, то в данном контексте `birdie` является уткой. По определению, утиная типизация вступает в действие только на этапе выполнения, когда предпринимается попытка выполнить какую-то операцию объекта. Это более гибкий подход, чем *номинальная типизация*, но расплачиваться за гибкость приходится большим числом ошибок на этапе выполнения[1].

Номинальная типизация

Этот взгляд принят в C++, Java и C# и поддерживается в Python с аннотациями. У объектов и переменных есть типы. Но объекты существуют только во время выполнения, а средство проверки типов имеет дело лишь с исходным кодом, в котором переменные (в т. ч. параметры) аннотированы типами. Если `Duck` – подкласс `Bird`, то экземпляр типа `Duck` можно присвоить параметру, аннотированному как `birdie: Bird`. Но в теле функции средство проверки типов считает вызов `birdie.quack()` недопустимым, потому что `birdie` номинально принадлежит типу `Bird`, а в этом классе нет метода `.quack()`. Не важно, что фактический аргумент на этапе выполнения будет иметь тип `Duck`, потому что номинальная типизация проверяется статически. Средство проверки не выполняет никакую часть программы, а только читает исходный код. Это более строгий подход, чем *утиная типизация*, но зато он позволяет отлавливать некоторые ошибки на более ранних стадиях конвейера сборки, иногда даже в процессе ввода кода в IDE.

[1] Утиная типизация – неявная форма *структурной типизации*, которая в Python ⩾ 3.8 поддерживается также благодаря добавлению `typing.Protocol`. Эта тема обсуждается ниже в данной главе – в разделе «Статические протоколы», а затем более подробно в главе 13.

Пример 8.4 – незамысловатая иллюстрация противопоставления утиной и номинальной типизаций, а также статической проверки типов и поведения на этапе выполнения[1].

Пример 8.4. birds.py

```
class Bird:
    pass

class Duck(Bird): ❶
    def quack(self):
        print('Quack!')

def alert(birdie): ❷
    birdie.quack()

def alert_duck(birdie: Duck) -> None: ❸
    birdie.quack()

def alert_bird(birdie: Bird) -> None: ❹
    birdie.quack()
```

❶ `Duck` – подкласс `Bird`.
❷ `alert` не имеет аннотаций типов, поэтому средство проверки типов ее игнорирует.
❸ `alert_duck` принимает один аргумент типа `Duck`.
❹ `alert_bird` принимает один аргумент типа `Bird`.

Проверка *birds.py* с помощью Муру вскрывает проблему:

```
…/birds/ $ mypy birds.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)
```

Проанализировав только исходный код, Муру увидела, что в `alert_bird` есть ошибка: в аннотации параметр `birdie` объявлен как имеющий тип `Bird`, но в теле функции вызывается `birdie.quack()`, хотя в классе `Bird` нет такого метода.

Теперь попробуем воспользоваться модулем `birds` в файле *daffy.py* (пример 8.5).

Пример 8.5. daffy.py

```
from birds import *
daffy = Duck()

alert(daffy)       ❶
alert_duck(daffy)  ❷
alert_bird(daffy)  ❸
```

❶ Допустимый вызов, потому что для `alert` нет аннотаций типов.
❷ Допустимый вызов, потому что `alert_duck` принимает аргумент типа `Duck`, а `daffy` как раз и имеет тип `Duck`.

[1] Наследованием часто злоупотребляют, и его применение трудно оправдать в примерах одновременно простых и реалистичных, поэтому примите данный пример с животными как элементарную иллюстрацию подтипов.

❸ Допустимый вызов, потому что `alert_bird` принимает аргумент типа `Bird`, а тип `daffy`, `Duck`, является подклассом `Bird`.

Применение Муру к *daffy.py* выдает ту же ошибку по поводу вызова `quack` в функции `alert_bird`, определенной в файле *birds.py*:

```
…/birds/ $ mypy daffy.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)
```

Но Муру видит проблему не в самом файле *daffy.py*: все три вызова функций не вызывают возражений.

Запустив скрипт *daffy.py*, мы получим:

```
…/birds/ $ python3 daffy.py
Quack!
Quack!
Quack!
```

Все работает! Пригодность утиной типизации подтверждена!

На этапе выполнения Python не обращает внимания на объявленные типы. Он пользуется только утиной типизацией. Муру сочла функцию `alert_bird` ошибочной, но ее вызов для `daffy` работает правильно. У многих питонистов это поначалу вызывает удивление: средство статической проверки типов иногда находит ошибки там, где, как мы знаем, все работает нормально.

Однако если по прошествии нескольких месяцев вас попросят расширить этот глупейший пример, то, возможно, вы скажете Муру спасибо. Взгляните на модуль *woody.py*, в котором также используется `birds` (пример 8.6).

Пример 8.6. woody.py

```
from birds import *

woody = Bird()
alert(woody)
alert_duck(woody)
alert_bird(woody)
```

При проверке *woody.py* Муру обнаруживает две ошибки:

```
…/birds/ $ mypy woody.py
birds.py:16: error: "Bird" has no attribute "quack"
woody.py:5: error: Argument 1 to "alert_duck" has incompatible type "Bird";
expected "Duck"
Found 2 errors in 2 files (checked 1 source file)
```

Первая ошибка находится в файле *birds.py*: вызов `birdie.quack()` в функции `alert_bird`, это мы уже видели. Вторая ошибка находится в файле *woody.py*: `woody` – экземпляр `Bird`, поэтому вызов `alert_duck(woody)` недопустим, т. е. эта функция требует аргумента типа `Duck`. Каждая утка `Duck` является птицей `Bird`, но не каждая птица `Bird` является уткой `Duck`.

На этапе выполнения оба вызова в *woody.py* приведут к ошибке. Последовательность ошибок хорошо видна в консольном сеансе в примере 8.7.

Пример 8.7. Ошибки во время выполнения и чем могла бы помочь Муру

```
>>> from birds import *
>>> woody = Bird()
>>> alert(woody) ❶
Traceback (most recent call last):
  ...
AttributeError: 'Bird' object has no attribute 'quack'
>>>
>>> alert_duck(woody) ❷
Traceback (most recent call last):
  ...
AttributeError: 'Bird' object has no attribute 'quack'
>>>
>>> alert_bird(woody) ❸
Traceback (most recent call last):
  ...
AttributeError: 'Bird' object has no attribute 'quack'
```

❶ Муру не смогла обнаружить эту ошибку, потому что для `alert` нет аннотаций типов.

❷ Муру сообщила об ошибке: `Argument 1 to "alert_duck" has incompatible type "Bird"; expected "Duck"`.

❸ Муру, начиная с примера 8.4, талдычит нам о том, что функция `alert_bird` написана неправильно: `«Bird» has no attribute»quack»`.

Этот небольшой эксперимент показывает, что к работе с утиной типизацией проще приступить и она обладает большей гибкостью, но возможность, что неподдерживаемые операции приведут к ошибке во время выполнения, остается. Номинальная типизация обнаруживает ошибки до начала выполнения, но иногда отвергает код, который работал бы правильно – как вызов `alert_bird(daffy)` в примере 8.5. Даже если иногда это работает, функция `alert_bird` названа неудачно: в ее теле требуется объект, который поддерживает метод `.quack()`, а `Bird` его не поддерживает.

В этом нехитром примере все функции занимали одну строку. Но в реальной программе они могут быть длиннее, могут передавать аргумент `birdie` другим функциям, а место создания аргумента `birdie` может отстоять от места вызова на много функций, поэтому трудно понять, что именно привело к ошибке во время выполнения. Средство проверки типов предотвращает многие ошибки такого рода на этапе выполнения.

Ценность аннотаций типов сомнительна в таких крохотных примерах, которые могут поместиться в книге. Но становится более очевидной по мере увеличения размера кодовой базы. Именно поэтому компании, владеющие миллионами строк кода на Python, например Dropbox, Google и Facebook, инвестировали в команды и инструменты. Их цель – поддержать внедрение аннотаций типов во всей компании и проверять типы в значительной и постоянно растущей части кодовой базы в конвейере непрерывной интеграции.

В этом разделе мы исследовали взаимосвязь типов и операций при утиной и номинальной типизациях, начав с простой функции `double()`, которую

не снабдили никакими аннотациями типов. Теперь поговорим о наиболее важных типах, используемых для аннотирования функций. Мы покажем хороший способ добавить аннотации типов к `double()`, когда дойдем до раздела «Статические протоколы». Но прежде познакомимся с дополнительными фундаментальными типами.

Типы, пригодные для использования в аннотациях

Практически любой тип Python можно использовать в аннотациях типов, но есть кое-какие ограничения и рекомендации. Кроме того, модуль `typing` ввел в обращение специальные конструкции, семантика которых иногда вызывает удивление.

В этом разделе рассматриваются все основные типы, которые можно использовать в аннотациях:

- `typing.Any`;
- простые типы и классы;
- `typing.Optional` и `typing.Union`;
- коллекции общего вида, включая кортежи и отображения;
- абстрактные базовые классы;
- обобщенные итерируемые объекты;
- параметризованные обобщенные типы и `TypeVar`;
- `typing.Protocol` – ключ к *статической утиной типизации*;
- `typing.Callable`;
- `typing.NoReturn` – красноречивое завершение этого перечня.

Мы рассмотрим все эти типы по очереди, начав со странного типа, который кажется бесполезным, но на самом деле крайне важен.

Тип Any

Краеугольным камнем системы постепенной типизации является тип `Any`, или *динамический тип*. Когда средство проверки типов видит нетипизированную функцию вида

```
def double(x):
    return x * 2
```

оно предполагает, что:

```
def double(x: Any) -> Any:
    return x * 2
```

Это означает, что аргумент `x` и возвращаемое значение могут быть любого типа, необязательно одного и того же. Считается, что тип `Any` поддерживает любую операцию.

Сравним типы `Any` и `object`. Рассмотрим такую сигнатуру:

```
def double(x: object) -> object:
```

Эта функция также принимает аргументы любого типа, потому что любой тип является *подтипом* `object`.

Однако средство проверки типов такую функцию отвергает:

```
def double(x: object) -> object:
    return x * 2
```

Проблема в том, что `object` не поддерживает операцию `__mul__`. Вот что сообщает Муру:

```
…/birds/ $ mypy double_object.py
double_object.py:2: error: Unsupported operand types for *
("object" and "int")
Found 1 error in 1 file (checked 1 source file)
```

Чем более общим является тип, тем уже его интерфейс, т. е. тем меньше операций он поддерживает. Класс `object` реализует меньше операций, чем `abc.Sequence`, тот – меньше операций, чем `abc.MutableSequence`, а этот, в свою очередь, меньше операций, чем `list`.

Но `Any` – необычный тип, который располагается и в самом верху, и в самом низу иерархии типов. Это одновременно и самый общий тип, т. е. аргумент `n: Any` принимает значения любого типа, и самый специализированный, т. е. поддерживает любую операцию. По крайней мере, именно так средство проверки типов воспринимает `Any`.

Разумеется, никакой тип не может поддерживать все возможные операции, поэтому использование `Any` лишает средство проверки типов возможности выполнить свою основную миссию: обнаруживать потенциально некорректные операции до того, как программа аварийно завершится с исключением во время выполнения.

«Является подтипом» и «совместим с»

Традиционная объектно-ориентированная номинальная система типов опирается на отношение «является подтипом». Если дан класс `T1` и подкласс `T2`, то `T2` *является подтипом* `T1`.

Рассмотрим такой код:

```
class T1:
    ...
class T2(T1):
    ...
def f1(p: T1) -> None:
    ...
o2 = T2()
f1(o2) # OK
```

Вызов `f1(o2)` – пример применения принципа подстановки Лисков (Liskov Substitution Principle – LSP). Барабара Лисков[1] на самом деле определяла отношение «является подтипом» в терминах поддерживаемых операций: если объект типа `T2` можно подставить вместо объекта типа `T1` и программа будет вести себя корректно, то `T2` является подтипом `T1`.

Если продолжить предыдущий пример, то следующий код является нарушением LSP:

```
def f2(p: T2) -> None:
    ...

o1 = T1()
f2(o1) # ошибка типизации
```

[1] Профессор MIT, проектировщик языков программирования, лауреат премии Тьюринга. Википедия: Barbara Liskov (*https://en.wikipedia.org/wiki/Barbara_Liskov*).

С точки зрения поддерживаемых операций, это вполне разумно: будучи подклассом, `T2` наследует и должен поддерживать все операции `T1`. Поэтому экземпляр `T2` можно использовать всюду, где ожидается экземпляр `T1`. Но обратное неверно: `T2` может реализовывать дополнительные методы, поэтому может статься, что `T1` допустимо использовать не во всех местах, где ожидается `T2`. Такое внимание к поддерживаемым операциям отражено в названии *подтипизация по поведению* (*https://en.wikipedia.org/wiki/Behavioral_subtyping*), которое также используется в качестве синонима LSP.

В системе постепенной типизации есть еще одно отношение: «совместим с», которое применимо везде, где применимо отношение «является подтипом», но имеет дополнительные правила, касающиеся `Any`, а именно:

1. Если даны тип `T1` и подтип `T2`, то `T2` *совместим с* `T1` (подстановка Лисков).
2. Любой тип *совместим с* `Any`: в качестве аргумента, объявленного с типом `Any`, можно передать объект любого типа.
3. `Any` *совместим с* любым типом: объект типа `Any` можно передать в качестве аргумента, объявленного с любым типом.

Для показанных выше определений объектов `o1` и `o2` ниже приведены примеры допустимого кода, иллюстрирующие правила 2 и 3:

```
def f3(p: Any) -> None:
    ...

o0 = object()
o1 = T1()
o2 = T2()

f3(o0) #
f3(o1) # все хорошо: правило 2
f3(o2) #

def f4(): # неявный возвращаемый тип: `Any`
    ...

o4 = f4() # выведенный тип: `Any`
f1(o4) #
f2(o4) # все хорошо: правило 3
f3(o4) #
```

В любой системе постепенной типизации необходим универсальный тип наподобие `Any`.

Слова «вывести» в контексте анализа типов – наукообразный синоним слова «угадать». Современные программы проверки типов в Python и других языках не требуют наличия аннотаций типов всюду, потому что могут вывести тип многих выражений. Например, если я напишу `x = len(s) * 10`, то программе проверки типов не нужно видеть явное локальное объявление, чтобы понять, что `x` имеет тип `int`, при условии что она может найти аннотации типов для встроенной функции `len`.

Теперь можно перейти к изучению остальных типов, используемых в аннотациях.

Простые типы и классы

Простые типы, такие как `int`, `float`, `str` и `bytes`, можно использовать в аннотациях типов напрямую. Конкретные классы из стандартной библиотеки, внешних пакетов или определенные пользователем – `FrenchDeck`, `Vector2d` или `Duck` – тоже можно использовать в аннотациях.

В аннотациях типов могут быть полезны и абстрактные базовые классы. Мы вернемся к ним при изучении типов коллекций и в разделе «Абстрактные базовые классы» ниже.

Для классов отношение «совместим с» определяется как «является подклассом»: подкласс *совместим со* всеми своими суперклассами.

Однако «практичность важнее чистоты», поэтому существует важное исключение, которое составляет предмет следующей врезки.

INT совместим с COMPLEX

Номинально не существует отношения подтипа между встроенными типами `int`, `float` и `complex`: все они являются прямыми подклассами `object`. Но в документе PEP 484 декларировано (*https://peps.python.org/pep-0484/#the-numeric-tower*), что тип `int` *совместим с* `float`, а `float` *совместим с* `complex`. Это имеет смысл на практике: `int` реализует все операции, которые реализует `float`, плюс дополнительные: `&`, `|`, `<<` и т. д. Итог: `int` *совместим с* `complex`. Если `i = 3`, то `i.real` равно `3`, а `i.imag` равно `0`.

Типы Optional и Union

В разделе «None в качестве значения по умолчанию» мы встречались со специальным типом `Optional`. Он решает проблему использования `None` в качестве значения по умолчанию, как в следующем примере из того же раздела:

```
from typing import Optional
def show_count(count: int, singular: str, plural: Optional[str] = None) -> str:
```

Конструкция `Optional[str]` фактически является сокращенной записью `Union[str, None]`, которая означает, что типом `plural` может быть `str` или `None`.

Улучшенный синтаксис Optional и Union в Python 3.10

Начиная с версии Python 3.10 мы можем писать `str | bytes` вместо `Union[str, bytes]`. Это короче и не нужно импортировать `Optional` и `Union` из модуля `typing`. Сравните старый и новый синтаксисы аннотаций типов для параметра `plural` функции `show_count`:

```
plural: Optional[str] = None # до
plural: str | None = None    # после
```

Оператор `|` можно также применять в сочетании с `isinstance` и `issubclass` для построения второго аргумента: `isinstance(x, int | str)`. Дополнительные сведения см. в документе PEP 604 «Complementary syntax for Union[]» (*https://peps.python.org/pep-0604/*).

Сигнатура встроенной функции `ord` – простой пример `Union`: она принимает `str` или `bytes` и возвращает `int`[1]:

[1] Точнее, `ord` принимает только объект типа `str` или `bytes`, для которого `len(s) == 1`. Но в настоящее время система типов не позволяет выразить такое ограничение.

```
def ord(c: Union[str, bytes]) -> int: ...
```

Ниже приведен пример функции, которая принимает `str`, но может вернуть `str` или `float`:

```
from typing import Union

def parse_token(token: str) -> Union[str, float]:
    try:
        return float(token)
    except ValueError:
        return token
```

По возможности избегайте создания функций, возвращающих типы `Union`, поскольку они возлагают дополнительную ответственность на пользователя – он должен проверять тип возвращенного значения во время выполнения, чтобы знать, что с ним делать. Но функция `parse_token` выше – разумный пример использования в контексте простого вычислителя выражений.

В разделе «Двухрежимный API» главы 4 мы видели функции, которые принимают аргументы типа `str` или `bytes`, но возвращают `str`, если аргумент имел тип `str`, и `bytes`, если он имел тип `bytes`. В таких случаях тип возвращаемого значения определяется типом входа, поэтому `Union` – не совсем подходящее решение. Чтобы правильно аннотировать подобные функции, нам нужна переменная-тип (см. раздел «Параметризованные обобщенные типы и TypeVar») или перегрузка – как будет показано в разделе «Перегруженные сигнатуры».

`Union[]` требует по меньшей мере двух типов. Вложенные типы `Union` дают такой же эффект, как линеаризованный `Union`. То есть аннотация типа

```
Union[A, B, Union[C, D, E]]
```

– то же самое, что

```
Union[A, B, C, D, E]
```

`Union` полезнее в сочетании с типами, не совместимыми между собой. Например, `Union[int, float]` избыточно, потому что `int` *совместим с* `float`. Если просто воспользоваться `float` для аннотирования параметра, то значения типа `int` тоже будут считаться допустимыми.

Обобщенные коллекции

Большинство коллекций в Python гетерогенны. Например, в список `list` можно поместить значения произвольных типов, необязательно одинаковых. Но на практике это не очень полезно: помещая элементы в коллекцию, мы, скорее всего, хотим впоследствии производить над ними какие-то операции, и обычно это означает, что у всех элементов должен быть хотя бы один общий метод[1].

[1] В языке ABC, оказавшем наибольшее влияние на начальный проект Python, списки могли содержать значения только одного типа, а именно того, которому принадлежал первый помещенный в список элемент.

При объявлении обобщенных типов можно указывать тип элементов, которые они могут обрабатывать.

Например, `list` можно параметризовать, ограничив типы содержащихся в списке элементов, как показано в примере 8.8.

Пример 8.8. Функция `tokenize` с аннотациями типов для Python ⩾ 3.9

```
def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

В версиях Python ⩾ 3.9 это означает, что `tokenize` возвращает список `list`, в котором каждый элемент имеет тип `str`.

Аннотации `stuff: list` и `stuff: list[Any]` означают одно и то же: `stuff` может быть списком объектов любого типа.

В версиях Python 3.8 и более ранних концепция та же, но кода нужно написать больше – см. врезку «Поддержка унаследованных типов и нерекомендуемые типы коллекций» ниже.

В документе PEP 585 «Type Hinting Generics In Standard Collections» (*https://peps.python.org/pep-0585/#implementation*) перечислены коллекции из стандартной библиотеки, принимающие аннотации обобщенных типов. В следующем списке приведены только те коллекции, которые принимают простейшую аннотацию вида `container[item]`:

```
list            collections.deque    abc.Sequence    abc.MutableSequence
set             abc.Container        abc.Set         abc.MutableSet
frozenset       abc.Collection
```

Типы кортежа и отображения поддерживают и более сложные аннотации типов, как будет показано в соответствующих разделах.

В версии Python 3.10 нет хорошего способа аннотировать `array.array`, который принимал бы во внимание аргумент конструктора `typecode`, определяющий, какие числа хранятся в массиве: целые или с плавающей точкой. Еще более трудная проблема – как проверить диапазон целочисленных типов, чтобы во время выполнения предотвратить исключение `OverflowError` при добавлении элементов в массивы. Например, `array` с `typecode='B'` может хранить только значения типа `int` от 0 до 255. В настоящее время статическая система типов в Python не умеет справляться с этой задачей.

Поддержка унаследованных типов и нерекомендуемые типы коллекций

(Если вы работаете только с версией Python 3.9 или более поздней, можете пропустить этот материал.)

В Python 3.7 и 3.8 необходимо импортировать модуль `__future__`, чтобы нотация `[]` работала со встроенными коллекциями, в частности `list` (см. пример 8.9).

Пример 8.9. Функция `tokenize` с аннотациями типов в Python ⩾ 3.7

```
from __future__ import annotations

def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

Импорт `__future__` не работает с версией Python 3.6 и более ранними.

В примере 8.10 показано, как аннотировать функцию `tokenize` способом, работающим с Python ⩾ 3.5.

Пример 8.10. Функция `tokenize` с аннотациями типов в Python ⩾ 3.5

```
from typing import List

def tokenize(text: str) -> List[str]:
    return text.upper().split()
```

Чтобы с чего-то начать поддержку аннотаций обобщенных типов, авторы PEP 484 создали десятки обобщенных типов в модуле `typing`. В табл. 8.1 показаны некоторые из них. Полный список см. в документации (*https://docs.python.org/3/library/ typing.html*).

Таблица 8.1. Некоторые типы коллекций и соответствующие им аннотации типов

Коллекция	Соответствующая аннотация типа
`list`	`typing.List`
`set`	`typing.Set`
`frozenset`	`typing.FrozenSet`
`collections.deque`	`typing.Deque`
`collections.abc.MutableSequence`	`typing.MutableSequence`
`collections.abc.Sequence`	`typing.Sequence`
`collections.abc.Set`	`typing.AbstractSet`
`collections.abc.MutableSet`	`typing.MutableSet`

Документом PEP 585 «Type Hinting Generics In Standard Collections» (*https://peps.python.org/pep-0585/*) был начат многолетний процесс улучшения аннотаций обобщенного типа с точки зрения удобства пользования. Он состоял из четырех шагов.

1. Включить в Python 3.7 предложение `from __future__ import annotations`, чтобы стандартные библиотечные классы можно было использовать как обобщенные с помощью нотации `list[str]`.
2. Сделать это поведение подразумеваемым по умолчанию в Python 3.9: теперь `list[str]` работает без импорта `future`.
3. Объявить нерекомендуемыми избыточные обобщенные типы из модуля `typing`[1]. Предупреждения о нерекомендуемости не будут выдаваться интерпретатором Python, потому что средства проверки типов должны по-

[1] Я и сам внес вклад в документацию по модулю `typing` – добавил десятки предупреждений о нерекомендуемости, когда под чутким руководством Гвидо ван Россума разбивал раздел «Содержимое модуля» на подразделы.

мечать нерекомендуемые типы, когда проверяемая программа рассчитана на версию Python 3.9 или более новую.

4. Удалить избыточные обобщенные типы из первой же версии Python, выпущенной спустя пять лет после выхода Python 3.9. При нынешнем темпе выхода версий это будет версия Python 3.14 или Python Pi.

Теперь посмотрим, как включать в аннотации обобщенные типы.

Типы кортежей

Существует три способа аннотировать тип кортежа:

- кортежи как записи;
- кортежи как записи с именованными полями;
- кортежи как неизменяемые последовательности.

Кортежи как записи

Если кортеж используется как запись, то возьмите встроенный тип `tuple` и объявите типы полей в квадратных скобках `[]`.

Например, если мы хотим принимать кортежи, содержащие название города, численность населения в нем и страну: `('Shanghai', 24.28, 'China')`, то аннотация типа должна выглядеть так: `tuple[str, float, str]`.

Рассмотрим функцию, которая принимает географические координаты и возвращает объект Geohash (*https://en.wikipedia.org/wiki/Geohash*), используемый следующим образом:

```
>>> shanghai = 31.2304, 121.4737
>>> geohash(shanghai)
'wtw3sjq6q'
```

В примере 8.11 показано, как функция `geohash` определяется с использованием пакета `geolib` из архива PyPI.

Пример 8.11. coordinates.py с функцией `geohash`

```
from geolib import geohash as gh # type: ignore ❶

PRECISION = 9

def geohash(lat_lon: tuple[float, float]) -> str: ❷
    return gh.encode(*lat_lon, PRECISION)
```

❶ Эта команда не дает Муру ругаться на то, что в пакете `geolib` нет аннотаций типов.

❷ Параметр `lat_lon` аннотирован типом `tuple` с двумя полями типа `float`.

Для версий Python < 3.9 импортируйте и используйте `typing.Tuple` в аннотациях типов. Такая практика не рекомендуется, но останется в стандартной библиотеке по крайней мере до 2024 года.

Кортежи как записи с аннотированными полями

Чтобы использовать в аннотациях кортеж, содержащий много полей, или конкретные типы кортежей, которые встречаются во многих местах кода, я настоятельно рекомендую взять тип `typing.NamedTuple`, описанный в главе 5. В примере 8.12 показан вариант примера 8.11 с `NamedTuple`.

Пример 8.12. coordinates_named.py с `NamedTupleCoordinates` и функцией `geohash`

```
from typing import NamedTuple

from geolib import geohash as gh # type: ignore

PRECISION = 9

class Coordinate(NamedTuple):
    lat: float
    lon: float

def geohash(lat_lon: Coordinate) -> str:
    return gh.encode(*lat_lon, PRECISION)
```

Как было объяснено в разделе «Обзор построителей классов данных» главы 5, класс `typing.NamedTuple` является фабрикой подклассов `tuple`, так что тип `Coordinate` *совместим с* `tuple[float, float]`, но обратное неверно – ведь у `Coordinate` есть дополнительные методы, добавленные `NamedTuple`, например `._asdict()`, и могут быть также определенные пользователем методы.

На практике это означает, что можно безопасно передать экземпляр `Coordinate` функции `display`, определенной следующим образом:

```
def display(lat_lon: tuple[float, float]) -> str:
    lat, lon = lat_lon
    ns = 'N' if lat >= 0 else 'S'
    ew = 'E' if lon >= 0 else 'W'
    return f'{abs(lat):0.1f}°{ns}, {abs(lon):0.1f}°{ew}'
```

Кортежи как неизменяемые последовательности

Чтобы включить в аннотацию кортеж неопределенной длины, используемый как неизменяемый список, необходимо задать один тип, за которым следуют запятая и многоточие `...` (это лексема Python, составленная из трех точек, а не символ Unicode `U+2026 — HORIZONTAL ELLIPSIS`).

Например, `tuple[int, ...]` – кортеж, состоящий из элементов типа `int`.

Многоточие означает, что допустимо любое число элементов >= 1. Не существует способа задать поля разных типов в кортежах произвольной длины.

Аннотации `stuff: tuple[Any, ...]` и `stuff: tuple` означают одно и то же: `stuff` является кортежем неопределенной длины, содержащим объекты любого типа.

В примере ниже функция `columnize` преобразует последовательность в таблицу, строки которой представляют собой список кортежей неопределенной длины. Это полезно, когда нужно отобразить элементы по столбцам:

```
>>> animals = 'drake fawn heron ibex koala lynx tahr xerus yak
zapus'.split()
>>> table = columnize(animals)
>>> table
```

```
[('drake', 'koala', 'yak'), ('fawn', 'lynx', 'zapus'), ('heron',
'tahr'), ('ibex', 'xerus')]
>>> for row in table:
...     print(''.join(f'{word:10}' for word in row))
...
drake     koala     yak
fawn      lynx      zapus
heron     tahr
ibex      xerus
```

В примере 8.13 показана реализация `columnize`. Обратите внимание на тип возвращаемого значения:

```
list[tuple[str, ...]]
```

Пример 8.13. columnize.py возвращает список кортежей строк

```
from collections.abc import Sequence

def columnize(
    sequence: Sequence[str], num_columns: int = 0
) -> list[tuple[str, ...]]:
    if num_columns == 0:
        num_columns = round(len(sequence) ** 0.5)
    num_rows, reminder = divmod(len(sequence), num_columns)
    num_rows += bool(reminder)
    return [tuple(sequence[i::num_rows]) for i in range(num_rows)]
```

Обобщенные отображения

Обобщенные отображения – это аннотированные типы вида `MappingType[KeyType,ValueType]`. Встроенный тип `dict` и типы отображений в модулях `collections` и `collections.abc` следуют соглашению, принятому в Python ⩾ 3.9. В более ранних версиях необходимо использовать класс `typing.Dict` и другие типы отображений из модуля `typing`, как описано во врезке «Поддержка унаследованных типов и нерекомендуемые типы коллекций» выше.

Пример 8.14 – практическая демонстрация использования функции, возвращающей инвертированный индекс (*https://en.wikipedia.org/wiki/ Inverted_index*) для поиска символов Unicode по имени – вариация на тему примера 4.21, лучше приспособленная для серверного кода, который мы будем изучать в главе 21.

Получив начальный и конечный коды символов Unicode, `name_index` возвращает словарь `dict[str, set[str]]`, который содержит инвертированный индекс, отображающий слово во множество символов, в названии которых это слово встречается. Например, после индексирования ASCII-символов от 32 до 64 слов `'SIGN'` и `'DIGIT'` будут соответствовать показанные ниже множества символов. Показано также, как найти символ с именем `'DIGIT EIGHT'`:

```
>>> index = name_index(32, 65)
>>> index['SIGN']
{'$', '>', '=', '+', '<', '%', '#'}
>>> index['DIGIT']
{'8', '5', '6', '2', '3', '0', '1', '4', '7', '9'}
>>> index['DIGIT'] & index['EIGHT']
{'8'}
```

В примере 8.14 приведен исходный код файла *charindex.py*, содержащего функцию `name_index`. Помимо аннотации типа `dict[]`, в этом примере демонстрируются еще три средства, которые до сих пор нам не встречались.

Пример 8.14. charindex.py

```
import sys
import re
import unicodedata

from collections.abc import Iterator

RE_WORD = re.compile(r'\w+')
STOP_CODE = sys.maxunicode + 1

def tokenize(text: str) -> Iterator[str]: ❶
    """вернуть итерируемый объект, содержащий слова в верхнем регистре"""
    for match in RE_WORD.finditer(text):
        yield match.group().upper()

def name_index(start: int = 32, end: int = STOP_CODE) -> dict[str, set[str]]:
    index: dict[str, set[str]] = {} ❷
    for char in (chr(i) for i in range(start, end)):
        if name := unicodedata.name(char, ''): ❸
            for word in tokenize(name):
                index.setdefault(word, set()).add(char)
    return index
```

❶ `tokenize` – генераторная функция. Генераторам посвящена глава 17.

❷ Локальная переменная `index` аннотирована. Не будь этой аннотации, Муру сообщила бы: `Need type annotation for 'index' (hint: «index: dict[<type>, <type>] = ...»)`.

❸ Я воспользовался оператором `:=` в условии `if`. Он присваивает результат вызова `unicodedata.name()` переменной `name` и выражению в целом. Результат `''` интерпретируется как ложь и `index` не изменяется[1].

Когда `dict` выступает в роли записи, принято делать все ключи объектами типа `str`, а тип значения может зависеть от ключа. Этот вопрос рассматривается в разделе «TypedDict» главы 15.

Абстрактные базовые классы

> Будь консервативен, когда передаешь, будь либерален, когда принимаешь.
>
> – Закон Постела, или принцип надежности

В табл. 8.1 перечислено несколько абстрактных классов из модуля `collections.abc`. В идеале функция должна принимать аргументы этих абстрактных типов (или их эквивалентов из модуля `typing` в версиях младше Python 3.9), а не конкретные типы. Это дает больше гибкости вызывающей стороне.

[1] Я использую оператор `:=` в нескольких примерах, когда это имеет смысл, но не рассматриваю его в данной книге. Технические детали см. в документе PEP 572 «Assignment Expressions» (*https://peps.python.org/pep-0572/*).

Рассмотрим следующую сигнатуру функции:

```
from collections.abc import Mapping

def name2hex(name: str, color_map: Mapping[str, int]) -> str:
```

Использование `abc.Mapping` позволяет вызывающей стороне передать экземпляр `dict`, `defaultdict`, `ChainMap`, подкласса `UserDict` или любого другого типа, *являющегося подтипом* `Mapping`.

С другой стороны, рассмотрим сигнатуру:

```
def name2hex(name: str, color_map: dict[str, int]) -> str:
```

Теперь `color_map` должно быть объектом типа `dict` или одного из его подтипов, например `defaultDict` или `OrderedDict`. Но подкласс `collections.UserDict` не пройдет проверку типа, хотя это и рекомендованный способ создавать пользовательские отображения, как мы видели в разделе «Создание подкласса UserDict вместо dict» главы 3. Муру отвергнет экземпляр класса `UserDict` или производного от него, потому что `UserDict` не является подклассом `dict`; они находятся на одном уровне иерархии наследования – оба являются подклассами `abc.MutableMapping`[1].

Таким образом, в общем случае в аннотациях типов параметров лучше использовать `abc.Mapping` или `abc.MutableMapping`, а не `dict` (или `typing.Dict` в унаследованном коде). Если функции `name2hex` не нужно изменять переданное `color_map`, то самой точной аннотацией типа `color_map` будет `abc.Mapping`. Тогда вызывающей стороне не нужно будет предоставлять объект, реализующий методы `setdefault`, `pop` и `update`, которые являются частью интерфейса `MutableMapping`, но не `Mapping`. Это вполне согласуется со второй частью закона Постела: «будь либерален, когда принимаешь».

Закон Постела также рекомендует быть консервативным при передаче. Значением функции всегда является конкретный объект, поэтому в аннотации для возвращаемого значения должен быть указан конкретный тип, как в примере из раздела «Обобщенные коллекции», где используется `list[str]`:

```
def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

В разделе документации, посвященном `typing.List` (*https://docs.python.org/3/library/typing.html#typing.List*), говорится:

> Обобщенная версия `list`. Полезна для аннотирования типов возвращаемых значений. Для аннотирования аргументов лучше использовать абстрактные типы коллекций, например `Sequence` или `Iterable`.

Аналогичный комментарий имеется в разделах, посвященных `typing.Dict` (*https://docs.python.org/3/library/typing.html#typing.Dict*) и `typing.Set` (*https://docs.python.org/3/ library/typing.html#typing.Set*).

Напомним, что начиная с Python 3.9 большинство ABC из модуля `collections.abc` и другие конкретные классы из модуля `collections`, а также встроенные

[1] На самом деле `dict` – это виртуальный подкласс `abc.MutableMapping`. Концепция виртуального подкласса объясняется в главе 13. Пока достаточно знать, что `issubclass(dict,abc.MutableMapping)` равно `True`, несмотря на то что `dict` написан на C и не наследует ничему из модуля `abc.MutableMapping`, а только `object`.

коллекции поддерживают нотацию аннотаций обобщенных классов вида `collections.deque[str]`. Соответствующие коллекции из модуля `typing` нужны только для поддержки кода, написанного для версии Python 3.8 или более ранней. Полный список классов, ставших обобщенными, приведен в разделе «Реализация» (*https://peps.python.org/pep-0585/#implementation*) документа PEP 585 «Type Hinting Generics In Standard Collections».

И, завершая обсуждение ABC в аннотациях типов, хотелось бы сказать несколько слов об ABC из пакета `numbers`.

Падение числовой башни

Пакет `numbers` определяет так называемую *числовую башню*, описанную в документе PEP 3141 «A Type Hierarchy for Numbers» (*https://peps.python.org/pep-3141/*). Башня представляет собой линейную иерархию ABC, наверху которой находится тип `Number`:

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

Эти ABC прекрасно работают для проверки типов во время выполнения, но не поддерживаются для статической проверки типов. В разделе «Числовая башня» (*https://peps.python.org/pep-0484/#the-numeric-tower*) документа PEP 484 ABC из пакета `numbers` отвергаются и постулируется, что встроенные типы `complex`, `float` и `int` должны считаться специальными случаями, как объясняется во врезке «INT совместим с COMPLEX».

Мы вернемся к этому вопросу в разделе «ABC из пакета numbers и числовые протоколы» главы 13, посвященном сравнению протоколов и ABC.

На практике, если требуется аннотировать числовые алгоритмы для статической проверки типов, у нас есть несколько вариантов.

1. Использовать один из конкретных типов `int`, `float` или `complex`, как рекомендуется в документе PEP 488.
2. Объявить тип объединения, например `Union[float, Decimal, Fraction]`.
3. Чтобы не зашивать в код конкретные типы, использовать числовые протоколы, например `SupportsFloat`, рассматриваемые в разделе «Статические протоколы, допускающие проверку во время выполнения» главы 13.

Раздел «Статические протоколы» ниже в этой главе – необходимое предварительное условие для понимания того, что такое числовые протоколы.

А пока обратимся к одному из самых полезных ABC для аннотаций типов: `Iterable`.

Тип Iterable

В документации по типу `typing.List`, которую я недавно цитировал, рекомендуется использовать типы `Sequence` и `Iterable` для аннотирования параметров функций. Пример аргумента типа `Iterable` встречается в функции `math.fsum` из стандартной библиотеки:

```
def fsum(__seq: Iterable[float]) -> float:
```

Файлы-заглушки и проект Typeshed

В версии Python 3.10 в стандартной библиотеке нет аннотаций типов, но Муру, PyCharm и другие программы могут найти необходимые аннотации в проекте Typeshed (*https://github.com/python/typeshed*) в виде *файлов-заглушек* – специальных исходных файлов с расширением *.pyi*, содержащих аннотированные сигнатуры функций и методов без реализации – очень похоже на заголовочные файлы в C.

Сигнатура функции `math.fsum` находится в файле */stdlib/2and3/math.pyi*. Начальные пробелы в имени `__seq` – соглашение о чисто позиционных параметрах, рекомендуемое в документе PEP 484 (мы объясним его в разделе «Аннотирование чисто позиционных и вариадических параметров» ниже).

Пример 8.15 – еще один пример использования параметра типа `Iterable`, который порождает элементы, имеющие тип `tuple[str, str]`. Вот как используется эта функция:

```
>>> l33t = [('a', '4'), ('e', '3'), ('i', '1'), ('o', '0')]
>>> text = 'mad skilled noob powned leet'
>>> from replacer import zip_replace
>>> zip_replace(text, l33t)
'm4d sk1ll3d n00b p0wn3d l33t'
```

А ниже показана ее реализация.

Пример 8.15. replacer.py

```
from collections.abc import Iterable

FromTo = tuple[str, str] ❶

def zip_replace(text: str, changes: Iterable[FromTo]) -> str: ❷
    for from_, to in changes:
        text = text.replace(from_, to)
    return text
```

❶ `FromTo` – псевдоним типа: я присвоил `FromTo` тип `tuple[str, str]`, чтобы сигнатуру `zip_replace` было проще читать.

❷ `changes` должен иметь тип `Iterable[FromTo]`; это то же самое, что `Iterable[tuple[str, str]]`, но короче и легче воспринимается.

Явные псевдонимы типов в Python 3.10

В документе PEP 613 «Explicit Type Aliases» (*https://peps.python.org/pep-0613/*) введен специальный тип `TypeAlias`, идея которого в том, чтобы сделать создаваемые псевдонимы типов хорошо видимыми и упростить для них проверку типов. Начиная с версии Python 3.10 это рекомендуемый способ создания псевдонимов типов:

```
from typing import TypeAlias

FromTo: TypeAlias = tuple[str, str]
```

abc.Iterable и abc.Sequence

И `math.fsum`, и `replacer.zip_replace` должны обойти все аргументы, порождаемые `Iterable`, чтобы вернуть результат. Учитывая, что бывают бесконечные итерируемые объекты, например генератор `itertools.cycle`, эти функции могли бы потребить всю память, что приведет к аварийному завершению процесса. Несмотря на эту потенциальную опасность, в современном Python довольно часто встречаются функции, которые принимают на входе `Iterable` даже тогда, когда для получения результата необходимо обработать его целиком. Это дает вызывающей стороне возможность поставлять входные данные с помощью генератора, а не строить последовательность заранее. Если количество входных элементов велико, то так можно сэкономить много памяти.

С другой стороны, функция `columnize` из примера 8.13 принимает параметр `Sequence`, а не `Iterable`, потому что ей нужно знать длину последовательности `len()`, чтобы заранее вычислить количество строк.

Как и `Sequence`, объект `Iterable` лучше использовать в качестве типа параметра. В качестве типа возвращаемого значения он не позволяет составить представление о том, что же будет на выходе. Функция должна более ясно говорить о том, какой конкретный тип она возвращает.

С `Iterable` тесно связан тип `Iterator`, используемый в качестве возвращаемого значения в примере 8.14. Мы вернемся к нему в главе 17, посвященной генераторам и классическим итераторам.

Параметризованные обобщенные типы и TypeVar

Параметризованный обобщенный тип – это обобщенный тип, записанный в виде `list[T]`, где `T` – переменная-тип, которая будет связана с конкретным типом при каждом использовании. Это позволяет использовать переменную-тип в типе результата.

В примере 8.16 определена функция `sample`, которая принимает два аргумента: последовательность элементов типа `T` и `int`. Она возвращает список `list` элементов того же типа `T`, случайно выбранных из первого аргумента.

Пример 8.16. sample.py

```
from collections.abc import Sequence
from random import shuffle
from typing import TypeVar

T = TypeVar('T')

def sample(population: Sequence[T], size: int) -> list[T]:
    if size < 1:
        raise ValueError('size must be >= 1')
    result = list(population)
    shuffle(result)
    return result[:size]
```

Ниже приведено два примера, показывающих, почему я использовал в `sample` переменную-тип:

- если при вызове функции передается кортеж типа `tuple[int, ...]`, который *совместим с* `Sequence[int]`, то параметр-тип равен `int`, т. е. возвращаемое значение будет иметь тип `list[int]`;

- если при вызове функции передается строка `str`, которая *совместима с* `Sequence[str]`, то параметр-тип равен `str`, т. е. возвращаемое значение будет иметь тип `list[str]`.

Зачем нужен класс TypeVar?

Авторы документа PEP 484 хотели ввести аннотации типов, добавив модуль `typing` и больше не изменяя в языке ничего. С помощью хитроумного метапрограммирования они могли бы заставить оператор `[]` работать с классами наподобие `Sequence[T]`. Но имя переменной `T` в квадратных скобках должно быть где-то определено, иначе в интерпретатор Python пришлось бы вносить глубокие изменения для поддержки нотации обобщенных типов как специального случая употребления `[]`. Поэтому понадобился конструктор `typing.TypeVar`: чтобы ввести имя переменной в текущее пространство имен. В языках типа Java, C# и TypeScript заранее объявлять имя переменной-типа не нужно, поэтому в них нет эквивалента класса `TypeVar`.

Еще один пример дает функция `statistics.mode` из стандартной библиотеки, которая возвращает самую часто встречающуюся точку в последовательности.

Вот пример ее использования из документации (*https://docs.python.org/3/library/statistics.html#statistics.mode*):

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

Без использования `TypeVar` функция `mode` могла бы иметь сигнатуру, показанную в примере 8.17.

Пример 8.17. mode_float.py: функция `mode`, применимая к `float` и его подтипам[1]

```
from collections import Counter
from collections.abc import Iterable

def mode(data: Iterable[float]) -> float:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError('no mode for empty data')
    return pairs[0][0]
```

Чаще всего `mode` применяется к значениям типа `int` или `float`, но в Python есть и другие числовые типы, и желательно, чтобы возвращалось значение того же типа, который указан во входном объекте `Iterable`. Мы можем улучшить сигнатуру, воспользовавшись классом `TypeVar`. Начнем с простой, но неправильной параметризованной сигнатуры:

```
from collections.abc import Iterable
from typing import TypeVar

T = TypeVar('T')

def mode(data: Iterable[T]) -> T:
```

[1] Здесь реализация проще, чем в модуле `statistics` из стандартной библиотеки Python.

При первом вхождении в сигнатуру параметр-тип `T` может быть любым. Но при втором вхождении он будет означать тот же тип, что при первом.

Поэтому любой итерируемый объект *совместим с* `Iterable[T]`, в т. ч. итерируемые объекты нехешируемых типов, которые не может обработать тип `collections.Counter`. Необходимо ограничить типы, которые можно присваивать `T`. В следующих двух разделах мы рассмотрим два способа это сделать.

Ограниченный TypeVar

Тип `TypeVar` принимает дополнительные позиционные аргументы, ограничивающие параметр-тип. Улучшить сигнатуру `mode`, так чтобы она принимала определенные числовые типы, можно следующим образом:

```
from collections.abc import Iterable
from decimal import Decimal
from fractions import Fraction
from typing import TypeVar

NumberT = TypeVar('NumberT', float, Decimal, Fraction)

def mode(data: Iterable[NumberT]) -> NumberT:
```

Уже лучше, именно такой была сигнатура `mode` в файле-заглушке *statistics.pyi* (*https://github.com/python/typeshed/blob/e1e99245bb46223928eba68d4fc74962240ba5b4/stdlib/3/statistics.pyi*) 25 мая 2020 года.

Однако в документацию по функции `statistics.mode` (*https://docs.python.org/3/library/statistics.html#statistics.mode*) включен такой пример:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

В спешке мы могли бы просто добавить `str` в определение `NumberT`:

```
NumberT = TypeVar('NumberT', float, Decimal, Fraction, str)
```

Это, конечно, работает, но теперь название типа `NumberT` совершенно не отражает его сущности, ведь он принимает `str`. Однако важнее даже не это, а то, что мы не можем добавлять типы всякий раз, как оказывается, что `mode` должна с ними работать. Есть способ лучше – нужно только воспользоваться другой возможностью `TypeVar`.

Связанный TypeVar

Взглянув на тело функции `mode` из примера 8.17, мы увидим, что для ранжирования используется класс `Counter`. Этот класс основан на словаре `dict`, поэтому тип каждого элемента, порождаемого итерируемым объектом, должен допускать хеширование.

На первый взгляд, должна работать следующая сигнатура:

```
from collections.abc import Iterable, Hashable

def mode(data: Iterable[Hashable]) -> Hashable:
```

Теперь проблема в том, что тип возвращаемого элемента `Hashable`: ABC, реализующий только метод `__hash__`. Поэтому средство проверки типов не позво-

лит нам сделать с возвращаемым значением ничего, кроме применения метода `hash()`. Не очень полезно.

Решение дает еще одна возможность `TypeVar`: необязательный именованный параметр `bound`. Он задает верхнюю границу допустимых типов. В примере 8.18 мы имеем `bound=Hashable`, это означает, что параметр-тип может быть `Hashable` или любым его подтипом[1].

Пример 8.18. mode_hashable.py: то же, что пример 8.17, но с более гибкой сигнатурой

```
from collections import Counter
from collections.abc import Iterable, Hashable
from typing import TypeVar

HashableT = TypeVar('HashableT', bound=Hashable)

def mode(data: Iterable[HashableT]) -> HashableT:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError('no mode for empty data')
    return pairs[0][0]
```

Подведем итоги:

- ограниченной переменной-типу будет назначен один из типов, перечисленных в объявлении `TypeVar`;
- связанной переменной-типу будет назначен выведенный тип выражения – при условии что выведенный тип *совместим с* границей, объявленной в именованном аргументе `bound=` типа `TypeVar`.

Не очень хорошо, что именованный аргумент для объявления граничного `TypeVar` называется `bound=`, потому что глагол «to bind» (связывать) обычно означает задание значения переменной, это действие в ссылочной семантике Python лучше всего описать как связывание имени со значением. Путаницы было бы меньше, если бы этот аргумент назывался `boundary=`.

У конструктора `typing.TypeVar` есть и другие факультативные параметры – `covariant` и `contravariant`, – которые мы будем обсуждать в разделе «Вариантность» главы 15.

Закончим это введение в тип `TypeVar` описанием переменной `AnyStr`.

Предопределенная переменная-тип AnyStr

В модуль `typing` включена переменная-тип `AnyStr`, определенная следующим образом:

```
AnyStr = TypeVar('AnyStr', bytes, str)
```

Она используется во многих функциях, принимающих `bytes` или `str` и возвращающих значение заданного типа.

А теперь перейдем к `typing.Protocol`, новой возможности, которая появилась в версии Python 3.8 и поддерживает использование аннотаций типов более органичным для Python способом.

---

[1] Я отправил это решение в `typeshed`, и именно так аннотирована функция `mode` в файле *statistics.pyi* по состоянию на 26 мая 2020 года.

Статические протоколы

В объектно-ориентированном программировании концепция «протокола» как неформального интерфейса восходит еще к Smalltalk и являлась неотъемлемой частью Python с самого начала. Однако в контексте аннотаций типов протокол – это подкласс `typing.Protocol`, определяющий интерфейс, который может верифицировать средство проверки типов. Оба вида протоколов рассматриваются в главе 13. Это лишь краткое введение в контексте аннотаций функций.

Тип `Protocol` в том виде, в каком он представлен в документе PEP 544 «Protocols: Structural subtyping (static duck typing)» (*https://peps.python.org/pep-0544/*), похож на интерфейсы в языке Go: тип протокола определяется заданием одного или нескольких методов, а средство проверки типов проверяет, что эти методы реализованы всюду, где требуется данный тип протокола.

В Python определение протокола записывается в виде подкласса `typing.Protocol`. Однако классы, *реализующие* протокол, не обязаны наследовать, регистрировать или объявлять какую-то связь с классом, который *определяет* протокол. На средство проверки типов возлагается обязанность найти имеющиеся типы протоколов и гарантировать их правильное использование.

Приведем пример задачи, которую можно решить с помощью `Protocol` и `TypeVar`. Предположим, что требуется создать функцию `top(it, n)`, которая возвращает *n* наибольших элементов из итерируемого объекта `it`:

```
>>> top([4, 1, 5, 2, 6, 7, 3], 3)
[7, 6, 5]
>>> l = 'mango pear apple kiwi banana'.split()
>>> top(l, 3)
['pear', 'mango', 'kiwi']
>>>
>>> l2 = [(len(s), s) for s in l]
>>> l2
[(5, 'mango'), (4, 'pear'), (5, 'apple'), (4, 'kiwi'), (6,
'banana')]
>>> top(l2, 3)
[(6, 'banana'), (5, 'mango'), (5, 'apple')]
```

Параметризованная обобщенная функция `top` могла бы выглядеть, как показано в примере 8.19.

Пример 8.19. Функция `top` с неопределенным параметром типом `T`

```
def top(series: Iterable[T], length: int) -> list[T]:
    ordered = sorted(series, reverse=True)
    return ordered[:length]
```

Вопрос в том, как ограничить `T`. Это не может быть `Any` или `object`, потому что `series` должен работать с функцией `sorted`. Встроенная функция `sorted` на самом деле принимает `Iterable[Any]`, но это только потому, что факультативный параметр `key` принимает функцию, которая вычисляет произвольный ключ сортировки по каждому элементу. А что будет, если передать `sorted` список простых объектов, но не задавать аргумент `key`? Попробуем:

```
>>> l = [object() for _ in range(4)]
>>> l
[<object object at 0x10fc2fca0>, <object object at 0x10fc2fbb0>,
<object object at 0x10fc2fbc0>, <object object at 0x10fc2fbd0>]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'object' and 'object'
```

Сообщение об ошибке говорит, что `sorted` применяет оператор `<` к элементам итерируемого объекта. И это все, что надо? Поставим еще один эксперимент[1]:

```
>>> class Spam:
...     def __init__(self, n): self.n = n
...     def __lt__(self, other): return self.n < other.n
...     def __repr__(self): return f'Spam({self.n})'
...
>>> l = [Spam(n) for n in range(5, 0, -1)]
>>> l
[Spam(5), Spam(4), Spam(3), Spam(2), Spam(1)]
>>> sorted(l)
   [Spam(1), Spam(2), Spam(3), Spam(4), Spam(5)]
```

Подтверждено: я могу сортировать список объектов `Spam`, потому что в классе `Spam` реализован специальный метод `__lt__`, поддерживающий оператор `<`.

Таким образом, параметр-тип `T` в примере 8.19 должен быть ограничен типами, реализующими `__lt__`. В примере 8.18 нам был нужен параметр-тип, реализующий метод `__hash__`, чтобы можно было использовать `typing.Hashable` как верхнюю границу типов. Но на этот раз ни в `typing`, ни в `abc` подходящего типа нет, так что придется его создать.

В примере 8.20 показан новый тип `SupportsLessThan`, являющийся протоколом.

Пример 8.20. comparable.py: определение типа протокола `SupportsLessThan`

```
from typing import Protocol, Any

class SupportsLessThan(Protocol): ❶
    def __lt__(self, other: Any) -> bool: ... ❷
```

❶ Протокол является подклассом `typing.Protocol`.

❷ В теле протокола имеется одно или несколько определений методов, содержащих вместо тела многоточие `...`.

Тип `T` *совместим с* протоколом `P`, если `T` реализует все методы, определенные в `P` с такими же сигнатурами.

Имея `SupportsLessThan`, мы теперь можем определить работоспособную версию функции `top`.

Пример 8.21. top.py: определение функции `top` с помощью `TypeVar` с `bound=SupportsLessThan`

```
from collections.abc import Iterable
from typing import TypeVar
```

[1] Как чудесно все-таки открыть интерактивную консоль и, опираясь на утиную типизацию, исследовать разные языковые средства. Мне очень не хватает этой возможности при работе с языками, которые ее не поддерживают.

```
from comparable import SupportsLessThan

LT = TypeVar('LT', bound=SupportsLessThan)

def top(series: Iterable[LT], length: int) -> list[LT]:
    ordered = sorted(series, reverse=True)
    return ordered[:length]
```

Протестируем `top`. В примере 8.22 показана часть комплекта тестов для `pytest`. Сначала мы пытаемся вызвать `top`, передав генераторное выражение, которое отдает `tuple[int, str]`, а затем – передав список `object`. Во втором случае мы ожидаем получить исключение `TypeError`.

Пример 8.22. top_test.py: часть комплекта тестов для `top`

```
from collections.abc import Iterator
from typing import TYPE_CHECKING ❶

import pytest

from top import top

# несколько строк опущено

def test_top_tuples() -> None:
    fruit = 'mango pear apple kiwi banana'.split()
    series: Iterator[tuple[int, str]] = ( ❷
        (len(s), s) for s in fruit)
    length = 3
    expected = [(6, 'banana'), (5, 'mango'), (5, 'apple')]
    result = top(series, length)
    if TYPE_CHECKING: ❸
        reveal_type(series) ❹
        reveal_type(expected)
        reveal_type(result)
    assert result == expected

# намеренно допущена ошибка типизации
def test_top_objects_error() -> None:
    series = [object() for _ in range(4)]
    if TYPE_CHECKING:
        reveal_type(series)
    with pytest.raises(TypeError) as excinfo:
        top(series, 3) ❺
    assert "'<' not supported" in str(excinfo.value)
```

❶ Константа `typing.TYPE_CHECKING` во время выполнения всегда равна `False`, но средства проверки типов во время работы притворяются, что она равна `True`.

❷ Явное объявление типа для переменной `series`, чтобы было проще читать вывод Муру[1].

❸ Это условие `if` запрещает выполнять следующие три строки во время прогона теста.

[1] Без этой аннотации типа Муру вывела бы для `series` тип `Generator[Tuple[builtins.int, builtins.str*], None, None]` – длинный, но совместимый с `Iterator[tuple[int, str]`.

❹ `reveal_type()` нельзя вызывать во время выполнения, потому что это не обычная функция, а отладочное средство Муру – потому-то оно ниоткуда не импортируется. Муру будет печатать по одному отладочному сообщению для каждого вызова псевдофункции `reveal_type()`, показывая, какой тип аргумента она вывела.

❺ Эту строку Муру пометит как ошибочную.

Показанные выше тесты проходят, но они прошли бы в любом случае – с аннотациями типов в *top.py* или без оных. Интереснее другое: если я проверю этот тестовый файл с помощью Муру, то увижу, что `TypeVar` работает в полном соответствии с ожиданиями. См. вывод `mypy` в примере 8.23.

В версии Муру 0.910 (июль 2021) `reveal_type` иногда показывает не в точности те типы, что я объявил, а совместимые с ними. Например, я использовал не `typing.Iterator`, а `abc.Iterator`. Не обращайте внимания на эту деталь. Вывод Муру все равно полезен. В ходе его обсуждения я сделаю вид, что эта проблема Муру уже решена.

Пример 8.23. Вывод mypy top_test.py

```
…/comparable/ $ mypy top_test.py
top_test.py:32: note:
    Revealed type is "typing.Iterator[Tuple[builtins.int, builtins.str]]" ❶
top_test.py:33: note:
    Revealed type is "builtins.list[Tuple[builtins.int, builtins.str]]"
top_test.py:34: note:
    Revealed type is "builtins.list[Tuple[builtins.int, builtins.str]]" ❷
top_test.py:41: note:
    Revealed type is "builtins.list[builtins.object*]" ❸
top_test.py:43: error:
    Value of type variable "LT" of "top" cannot be "object" ❹
Found 1 error in 1 file (checked 1 source file)
```

❶ В функции `test_top_tuples` `reveal_type(series)` показывает, что это `Iterator[tuple[int, str]]` – тот тип, который я явно объявил.

❷ `reveal_type(result)` подтверждает, что тип, возвращенный после вызова `top`, – именно то, что я хотел: с учетом типа `series` результат имеет тип `list[tuple[int, str]]`.

❸ В функции `test_top_objects_error` `reveal_type(series)` показывает, что это `list[object*]`. Муру выводит `*` после любого выведенного типа: я не аннотировал тип `series` в этом тесте.

❹ Муру обнаруживает ошибку, которую этот тест намеренно хотел вызвать: типом элемента итерируемого объекта не может быть `object` (элемент должен иметь тип `SupportsLessThan`).

Главное преимущество типа протокола перед ABC – то, что для совместимости с типом протокола не нужно никакого специального объявления. Это позволяет создавать протоколы с использованием уже существующих типов или типов, реализованных в коде, который мы не контролируем. Мне не нужно наследовать или регистрировать типы `str`, `tuple`, `float`, `set` и другие, чтобы использовать их там, где ожидается параметр типа `SupportsLessThan`. Они должны только реализовывать метод `__lt__`. А средство проверки все равно справится со своей

работой, потому что `SupportsLessThan` явно определен как `Protocol` – в противоположность неявным протоколам, свойственным утиной типизации, которые не видны средству проверки типов.

В документе PEP 544 «Protocols: Structural subtyping (static duck typing)» (*https://peps.python.org/pep-0544/*) был представлен специальный класс `Protocol`. В примере 8.21 демонстрируется, почему это средство называется *статическая утиная типизация*: решение аннотировать параметр `series` функции `top` было продиктовано следующим соображением: «Номинальный тип `series` не играет роли, лишь бы он реализовывал метод `__lt__`». Утиная типизация в Python всегда позволяла сказать это неявно, оставив средства статической проверки типов ни с чем. Средство проверки не может прочитать исходный код CPython, написанный на C, или выполнить эксперименты на консоли, дабы понять, что функция `sorted` требует лишь, чтобы элементы поддерживали оператор `<`.

Теперь же мы можем сделать утиную типизацию явной для программ статической проверки типов. Вот почему имеет смысл говорить, что `typing.Protocol` дает нам *статическую утиную типизацию*[1].

Это не все, что можно сказать о классе `typing.Protocol`. Мы вернемся к нему в части IV, где в главе 13 сравниваются структурная типизация, утиная типизация и ABC – еще один подход к формализации протоколов. Кроме того, в разделе «Перегруженные сигнатуры» главы 15 объясняется, как объявлять перегруженные сигнатуры функций с помощью декоратора `@typing.overload`, и приводится расширенный пример использования `typing.Protocol` и граничного `TypeVar`.

`typing.Protocol` открывает возможность для аннотирования функции `double`, представленной в разделе «Типы определяются тем, какие операции они поддерживают» выше, без потери функциональности. Ключ к этому – определение протокольного класса с методом `__mul__`. Предлагаю вам эту задачу в качестве упражнения. Решение будет приведено в разделе «Типизированная функция `double`» главы 13.

Тип Callable

Чтобы можно было аннотировать параметры обратного вызова или вызываемые объекты, возвращаемые функциями высшего порядка, модуль `collections.abc` предоставляет тип `Callable`, доступный также в модуле `typing` для тех, кто еще не перешел на Python 3.9. Тип `Callable` параметризован следующим образом:

```
Callable[[ParamType1, ParamType2], ReturnType]
```

Список параметров – `[ParamType1, ParamType2]` – может содержать нуль или более типов.

Ниже приведен пример в контексте функции `repl`, входящей в простой интерактивный интерпретатор, который мы разработаем в разделе «Сопоставление с образцом в lis.py: пример»[2].

[1] Я не знаю, кто придумал термин *статическая утиная типизация*, но он приобрел популярность с распространением языка Go, в котором семантика интерфейсов больше походит на протоколы Python, чем на номинальные интерфейсы Java.

[2] REPL означает Read-Eval-Print-Loop (цикл чтения-вычисления-печати), такой цикл лежит в основе интерактивных интерпретаторов.

```
def repl(input_fn: Callable[[Any], str] = input]) -> None:
```

При нормальном использовании функция `repl` пользуется встроенной в Python функцией `input` для чтения выражений, введенных пользователем. Однако для автоматизированного тестирования или интеграции с другими источниками ввода `repl` принимает факультативный параметр `input_fn`: объект типа `Callable` с такими же типами параметра и возвращаемого значения, что у `input`.

Встроенная функция `input` имеет следующую сигнатуру в typeshed:

```
def input(__prompt: Any = ...) -> str: ...
```

Сигнатура `input` *совместима с* таким аннотированным `Callable`:

```
Callable[[Any], str]
```

Не существует синтаксической конструкции, позволяющей аннотировать типы факультативных или именованных аргументов. В документации (*https://docs.python.org/3/library/typing.html#typing.Callable*) по `typing.Callable` сказано: «такие типы функций редко используются в качестве типов обратных вызовов». Если вам нужна аннотация типа для функции с гибкой сигнатурой, замените весь список параметров многоточием:

```
Callable[..., ReturnType]
```

Такое взаимодействие обобщенных параметров-типов с иерархией типов приводит к новой концепции: вариантности.

Вариантность в типах Callable

Представьте себе систему управления температурой с простой функцией `update`, показанной в примере 8.24. Функция `update` вызывает функцию `probe`, чтобы получить текущую температуру, а затем функцию `display`, чтобы показать температуру пользователю. Обе функции, `probe` и `display`, передаются `update` в качестве аргументов из педагогических соображений. Цель примера – сравнить две аннотации `Callable`: одну с типом возвращаемого значения, другую с типом параметра.

Пример 8.24. Иллюстрация вариантности

```
from collections.abc import Callable

def update( ❶
        probe: Callable[[], float], ❷
        display: Callable[[float], None] ❸
    ) -> None:
    temperature = probe()
    # здесь может быть какой-то код управления
    display(temperature)

def probe_ok() -> int: ❹
    return 42

def display_wrong(temperature: int) -> None: ❺
    print(hex(temperature))

update(probe_ok, display_wrong) # ошибка типизации ❻
```

```
def display_ok(temperature: complex) -> None: ❼
    print(temperature)

update(probe_ok, display_ok) # OK ❽
```

❶ `update` принимает два вызываемых объекта в качестве аргументов.
❷ Вызываемый объект `probe` не принимает аргументов и возвращает `float`.
❸ Объект `display` принимает аргумент типа `float` и возвращает `None`.
❹ Функция `probe_ok` *совместима с* `Callable[[], float]`, потому что возврат `int` не вызывает ошибки в коде, который ожидает `float`.
❺ Функция `display_wrong` *несовместима с* `Callable[[float], None]`, потому что нет гарантии, что функция, ожидающая `int`, сможет обработать `float`; например, функция Python `hex` принимает `int`, но отвергает `float`.
❻ Муру считает эту строку ошибочной, потому что `display_wrong` несовместима с аннотацией типа параметра `display` в `update`.
❼ `display_ok` *совместима с* `Callable[[float], None]`, потому что функция, принимающая `complex`, сможет обработать и `float`.
❽ Эта строка не вызывает возражений у Муру.

Подведем итоги. Ничто не мешает передать функцию обратного вызова, которая возвращает `int`, когда программа ожидает получить функцию, возвращающую `float`, потому что значение типа `int` всегда можно использовать, когда ожидается `float`.

Формально мы говорим, что `Callable[[], int]` *является подтипом* `Callable[[], float]`, – так как `int` *является подтипом* `float`. Это означает, что тип `Callable` *ковариантен* относительно возвращаемого значения, потому что направление отношения *является подтипом* между типами `int` и `float` такое же, как между типами `Callable`, в которых они являются типами возвращаемого значения.

С другой стороны, ошибкой будет передача функции обратного вызова, которая принимает аргумент типа `int`, когда требуется, чтобы функция могла обрабатывать `float`.

Формально `Callable[[int], None]` не *является подтипом* `Callable[[float], None]`. Хотя `int` *является подтипом* `float`, в параметризованном типе `Callable` направление отношения противоположно: `Callable[[float], None]` *является подтипом* `Callable[[int], None]`. Поэтому мы говорим, что тип `Callable` *контравариантен* относительно объявленных типов параметров.

В разделе «Вариантность» главы 15 эта тема рассматривается более подробно с примерами инвариантных, ковариантных и контравариантных типов.

Пока что имейте в виду, что большинство параметризованных обобщенных типов *инвариантны*, поэтому проще. Например, если я объявлю `scores: list[float]`, то буду точно знать, что можно присвоить `scores`. Я не смогу присвоить этой переменной объект, объявленный как `list[int]` или `list[complex]`:

- объект `list[int]` недопустим, потому что он может хранить значения типа `float`, которые моя программа могла бы поместить в `scores`;
- объект `list[complex]` недопустим, потому что моя программа может захотеть отсортировать `scores` для нахождения медианы, но тип `complex` не предоставляет метода `__lt__`, поэтому список `list[complex]` не сортируется.

Теперь рассмотрим последний специальный тип в этой главе.

Тип NoReturn

Этот специальный тип используется только для аннотирования типов возвращаемых значений функций, которые вообще не возвращают управления. Обычно они существуют, чтобы возбуждать исключение. В стандартной библиотеке десятки таких функций.

Например, функция `sys.exit()` возбуждает исключение `SystemExit`, чтобы завершить процесс Python.

Ее сигнатура в `typeshed` имеет вид:

```
def exit(__status: object = ...) -> NoReturn: ...
```

Параметр `__status` чисто позиционный и имеет значение по умолчанию. В файлах-заглушках не указываются значения по умолчанию, вместо них используется `...`. `__status` имеет тип `object`, а это значит, что он может принимать и значение `None`, поэтому было бы избыточно аннотировать его как `Optional[object]`.

В примере 24.6 главы 24 тип `NoReturn` используется в методе `__flag_unknown_attrs`, который выводит полное и понятное пользователю сообщение об ошибке, а затем возбуждает исключение `AttributeError`.

Последний раздел этой главы эпического размера посвящен позиционным и вариадическим параметрам.

Аннотирование чисто позиционных и вариадических параметров

Вспомним функцию `tag` из примера 7.9. Последний раз мы видели ее сигнатуру в разделе «Чисто позиционные параметры».

```
def tag(name, /, *content, class_=None, **attrs):
```

Ниже показана полностью аннотированная функция `tag`, записанная в нескольких строках – обычное соглашение для длинных сигнатур. Разбиение на строки такое, как в форматере *blue* (*https://pypi.org/project/blue/*):

```
from typing import Optional

def tag(
    name: str,
    /,
    *content: str,
    class_: Optional[str] = None,
    **attrs: str,
) -> str:
```

Обратите внимание на аннотацию типа `*content: str` для произвольных позиционных параметров; это означает, что все такие аргументы должны иметь `str`. Локальная переменная `content` в теле функции будет иметь тип `tuple[str, ...]`.

Произвольные именованные аргументы в этом примере аннотированы типом `**attrs: str`, поэтому аргумент `attrs` внутри функции будет иметь тип `dict[str, str]`. Если бы аннотация имела вид `**attrs: float`, то аргумент `attrs` имел бы тип `dict[str, float]`.

Если параметр `attrs` должен принимать значения разных типов, то следует использовать `Union[]` или `Any: **attrs: Any`.

Нотация `/` для чисто позиционных параметров появилась в версии Python 3.8. В Python 3.7 и более ранних версиях она приводит к синтаксической ошибке. В документе PEP 484 предлагается соглашение (*https://peps.python.org/pep-0484/#id38*): имя любого чисто позиционного параметра начинать с двух знаков подчеркивания. Снова приведем сигнатуру `tag`, на этот раз занимающую две строчки и следующую соглашению PEP 484:

```
from typing import Optional

def tag(__name: str, *content: str, class_: Optional[str] = None,
        **attrs: str) -> str:
```

Муру понимает оба способа объявления чисто позиционных параметров и следит за их соблюдением.

В заключение этой главы кратко рассмотрим ограничения аннотаций типов и статической системы типов, которую они поддерживают.

Несовершенная типизация и строгое тестирование

Лица, отвечающие за сопровождение больших корпоративных кодовых баз, говорят, что многие ошибки проще обнаружить и исправить с помощью программ статической проверки типов, чем если бы они были найдены только после запуска системы в эксплуатацию. Однако следует отметить, что в известных мне компаниях автоматизированное тестирование стало стандартной практикой и получило широкое распространение задолго до внедрения статической типизации.

Даже в тех контекстах, где статическая типизация наиболее полезна, на нее нельзя полагаться как на единственный критерий правильности. Вполне можно столкнуться с:

Ложноположительными результатами

Инструмент сообщает об ошибке, когда ее нет.

Ложноотрицательными результатами

Инструмент не сообщает об ошибке в заведомо неправильном коде.

Кроме того, требуя проверять типы всего на свете, мы утрачиваем часть выразительности Python:

- некоторые удобные возможности нельзя проверить статически, например распаковку аргументов вида `config(**settings)`;
- программы проверки типов в общем случае плохо поддерживают или вообще не понимают таких продвинутых возможностей, как свойства, дескрипторы, метаклассы и метапрограммирование;
- программы проверки типов не поспевают за выходом новых версий Python, отвергают код, содержащий новые возможности, или даже «падают» – и иногда это продолжается больше года.

Самые простые ограничения на данные невозможно выразить в системе типов. Например, аннотации типов не могут гарантировать, что «величина должна быть целым числом > 0» или что «метка должна быть строкой длиной

от 6 до 12 символов ASCII». Вообще, аннотации слабо помогают отлавливать ошибки в прикладной логике.

С учетом всех этих подводных камней аннотации типов не могут считаться оплотом качества программного обеспечения, а требовать их задания во всех случаях без исключения значило бы усугубить их недостатки.

Рассматривайте программы статической проверки типов как один из инструментов в современном конвейере непрерывной интеграции (CI) наряду с исполнителями тестов, линтерами и т. д. Цель конвейера CI – уменьшить количество программных ошибок, а автоматизированные тесты находят многие ошибки, выходящие за рамки возможного для аннотаций типов. Любой код, который можно написать на Python, можно на Python и протестировать – с аннотациями типов или без них.

Заголовок и заключение этого раздела были навеяны статьей Брюса Эккеля «Строгая типизация и строгое тестирование» (*https://docs.google.com/document/d/1aXs1tpwzPjW9MdsG5dI7clNFyYayFBkcXwRDo-qvbIk/preview*), опубликованной также в антологии «The Best Software Writing I» (*https://www.oreilly.com/library/view/the-best-software/ 9781590595008/*) под редакцией Джоэля Спольски (издательство Apress). Брюс – фанат Python и автор книг о C++, Java, Scala и Kotlin. В этой статье он рассказывает о том, что был горячим сторонником статической типизации, пока не изучил Python, и заключает следующими словами: «Если для программы на Python имеются адекватные автономные тесты, то она может быть не менее надежной, чем программы на C++, Java или C# с адекватным набором автономных тестов (хотя на Python тесты писать быстрее)».

На этом мы пока завершаем рассмотрение аннотаций типов в Python. Они также являются главной темой главы 15, где рассматриваются обобщенные классы, вариантность, перегруженные сигнатуры, приведение типов и т. д. А до тех пор аннотации типов будут время от времени появляться в примерах кода.

Резюме

Мы начали с краткого введения в концепцию постепенной типизации, а затем перешли к практическим вопросам. Трудно понять, как постепенная типизация работает, не имея инструмента, который читает аннотации типов, поэтому мы разработали аннотированную функцию, руководствуясь сообщениями об ошибках, выдаваемыми Муру.

Возвращаясь к идее постепенной типизации, мы выяснили, что она является гибридом традиционной утиной типизации в Python и номинальной типизации, более знакомой пишущим на Java, C++ и других статически типизированных языках.

Большая часть главы была посвящена представлению основных групп типов, используемых в аннотациях. Многие из рассмотренных типов тесно связаны со знакомыми нам типами объектов в Python: коллекциями, кортежами и вызываемыми объектами, но дополнены поддержкой нотации обобщения вида `Sequence[float]`. Многие из этих типов являются временными суррогатами,

реализованными в модуле `typing` до того, как стандартные типы были изменены с целью поддержки обобщенных типов в Python 3.9.

Некоторые типы являются специальными. Типы `Any`, `Optional`, `Union` и `NoReturn` не имеют никакого отношения к фактическим объектам в памяти, а существуют только в абстрактной сфере системы типов.

Мы изучили параметризованные обобщенные типы и переменные-типы, которые повышают гибкость аннотаций типов, не жертвуя типобезопасностью.

Параметризованные обобщенные типы становятся еще более выразительными при использовании протоколов. Тип `Protocol` появился только в версии Python 3.8 и пока еще не получил широкого распространения, но его важность невозможно переоценить. `Protocol` открывает возможность для статической утиной типизации, мосту между ядром Python, основанным на утиной типизации, и номинальной типизацией, позволяющей средствам статической проверки типов отлавливать ошибки.

По ходу изучения этих типов мы экспериментировали с программой Муру, чтобы понаблюдать за ошибками типизации и тем, какие типы выводит Муру, – с помощью магической функции `reveal_type()`.

Последний раздел был посвящен аннотированию чисто позиционных и вариадических параметров.

Аннотации типов – сложный и развивающийся предмет. По счастью, это факультативное средство. Давайте позаботимся о том, чтобы Python оставался доступным широчайшей аудитории, и не будем молиться, чтобы весь код на Python был снабжен аннотациями типов, – а я встречал публичные проповеди, произносимые поборниками типизации.

Наш почетный BDFL[1] возглавил проникновение аннотаций типов в Python, поэтому будет справедливо завершить эту главу его словами:

> Мне была бы не по нраву версия Python, в которой я был бы морально обязан всюду добавлять аннотации типов. Я действительно думаю, что у аннотаций типов есть свое место, но также полно случаев, когда их использование неуместно, поэтому хорошо, что у нас есть выбор[2].
>
> – Гвидо ван Россум

Дополнительная литература

Бернат Габор написал в своем замечательном посте «The state of type hints in Python» (*https://bernat.tech/posts/the-state-of-type-hints-in-python/*):

> Аннотации типов следовало бы использовать всюду, где имеет смысл писать автономные тесты.

Я сам большой поклонник тестирования, но при этом часто занимаюсь исследовательским кодированием. Во время экспериментов тесты и аннотации типов не особенно полезны. Они только тормозят работу.

---

[1] «Benevolent Dictator For Life» (пожизненный великодушный диктатор). См. статью Гвидо ван Россума «Origin of BDFL» (*https://www.artima.com/weblogs/viewpost.jsp?thread=235725*).

[2] Из видео на YouTube «Type Hints» (*https://www.youtube.com/watch?v=YFexUDjHO6w*) Гвидо ван Россума (март 2015). Цитата начинается в 13'40''. Я ее немного отредактировал.

Пост Габора – одно из лучших известных мне введений в аннотации типов в Python наряду со статьей Гейра Арне Хьелле «Python Type Checking (Guide)» (*https://realpython.com/python-type-checking/*). Текст Клаудио Йоловица «Hypermodern Python Chapter 4: Typing» (*https://cjolowicz.github.io/posts/hypermodern-python-04-typing/*) – более краткое введение, в котором рассматривается также проверка типов во время выполнения.

Если вам нужно более глубокое изложение, то наиболее авторитетным источником является документация по Муру (*https://mypy.readthedocs.io/en/stable/index.html*). Она представляет ценность независимо от того, каким средством проверки типов вы пользуетесь, потому что включает пособие и справочные материалы о типизации в Python вообще, а не только в самой программе Муру. Там же вы найдете удобные шпаргалки (*https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html*) и очень полезную страницу на тему типичных проблем и их решений (*https://mypy.readthedocs.io/en/stable/common_issues.html*).

Документация по модулю `typing` (*https://docs.python.org/3/library/typing.html*) – неплохой краткий справочник, но деталей там не слишком много. Документ PEP 483 «The Theory of Type Hints» (*https://peps.python.org/pep-0483/*) включает глубокое объяснение вариантности, причем тип `Callable` используется для иллюстрации контравариантности. Бесспорным авторитетом являются документы PEP, относящиеся к типизации. Их уже набралось больше 20. Предполагаемая аудитория PEP – разработчики ядра Python и Руководящий совет по Python, поэтому предполагаются обширные предварительные знания, и, конечно, это не легкое чтение.

Как уже было сказано, в главе 15 рассматриваются дополнительные вопросы типизации, там же, в разделе «Дополнительная литература», имеются другие ссылки, в т. ч. табл. 15.1, в которой перечислены PEP, относящиеся к типизации, – как утвержденные, так и находящиеся в процессе обсуждения (по состоянию на конец 2021 года).

Сайт «Awesome Python Typing» (*https://github.com/typeddjango/awesome-python-typing*) – ценное собрание ссылок на инструменты и справочные материалы.

Поговорим

Просто езжай

> Забудьте про ультралегкие неудобные велосипеды, блестящие майки, грубые ботинки, с трудом помещающиеся на крохотные педали, наматывание бесконечных миль. А просто езжайте, как бывало в детстве, – садитесь на велосипед и получайте чистую радость от катания.
>
> – Грант Петерсен «Просто езжай: радикально практичное руководство по катанию на велосипеде» (Workman Publishing)

Если кодирование – не ваша профессия, а лишь полезный инструмент, применяемый в профессиональных занятиях, или нечто такое, что вы изучаете ради удовольствия, то, пожалуй, аннотации типов нужны вам не больше, чем жесткие ботинки и металлические набойки большинству велосипедистов.

Просто кодируйте.

Когнитивный эффект типизации

Меня беспокоит, какое влияние оказывают аннотации типов на стиль кодирования на Python.

Я согласен, что пользователи большинства API выигрывают от наличия аннотаций типов. Но Python привлекал меня – среди многих других причин – тем, что предоставляет чрезвычайно мощные функции, способные заменить целые API, и, более того, мы сами можем писать такие функции. Рассмотрим встроенную функцию `max()`. Она мощная, но понять ее легко. Однако, как будет показано в разделе «Перегрузка max», чтобы ее правильно аннотировать, нужно 14 строчек аннотаций типов, и это не считая `typing.Protocol` и нескольких определений `TypeVar`, необходимых для поддержки этих аннотаций.

Я боюсь, что если библиотеки будут строго требовать аннотирования типов, то программисты раз и навсегда зарекутся писать такие функции.

Согласно англоязычной Википедии, «лингвистическая относительность» (*https://en.wikipedia.org/wiki/Linguistic_relativity*) – она же гипотеза Сепира–Уорфа – это «принцип, провозглашающий, что структура языка оказывает влияние на восприятие мира людьми, говорящими на нем». Далее Википедия объясняет:

- *сильная* версия говорит, что язык *определяет* мышление и что лингвистические категории ограничивают и определяют когнитивные категории;
- *слабая* версия говорит, что лингвистические категории и их использование лишь *влияют* на мышление и принимаемые решения.

В общем и целом лингвисты соглашаются, что сильная версия ложна, но существуют эмпирические свидетельства в поддержку слабой версии.

Мне неизвестны конкретные исследования в отношении языков программирования, но мой опыт показывает, что они сильно влияют на подход к проблеме. Первым языком, на котором я работал профессионально, был Applesoft BASIC – еще во времена 8-разрядных компьютеров. BASIC не поддерживал рекурсию непосредственно – приходилось самостоятельно раскручивать стек вызовов. Поэтому я никогда не рассматривал рекурсивные алгоритмы и структуры данных. Я знал, что на некотором концептуальном уровне такие вещи существовали, но они не входили в мой инструментарий.

Несколько десятков лет спустя, начиная осваивать Elixir, я получал наслаждение от рекурсивного решения задач и даже злоупотреблял рекурсией – пока не обнаружил, что многие мои решения были бы проще, если бы я использовал уже готовые функции из модулей Elixir `Enum` и `Stream`. Я узнал, что в идиоматическом коде прикладного уровня на Elixir рекурсивные вызовы редко бывают необходимы, зато перечисления и потоки реализуют рекурсию под капотом.

Гипотеза лингвистической относительности могла бы объяснить широко распространенную (хотя и не доказанную) идею о том, что изучение разных языков программирования делает из вас лучшего программиста, особенно когда языки поддерживают различные парадигмы программирования. Знание Elixir позволило мне применять функциональные паттерны при программировании на Python или Go.

А теперь вернемся на грешную землю.

Пакет `requests`, вероятно, имел бы совершенно другой API, если бы Кеннет Рейц решил сам (или ему велел бы начальник) аннотировать все функции. Он ставил себе целью написать API, которым было бы легко пользоваться, который был бы мощным и вместе с тем гибким. В этом он преуспел, и пакет `requests` снискал огромную популярность – в мае 2020 года он занимал четвертое место в статистике PyPI (*https://pypistats.org/top*) с числом скачиваний 2,6 миллиона в день. Первое место занимает пакет `urllib3`, от которого `requests` зависит.

В 2017 году люди, отвечающие за сопровождение `requests`, решили (*https://github.com/psf/requests/issues/3855*) не тратить время на написание аннотаций типов. Один из них, Кори Бенфилд, в почтовом сообщении писал:

> Я думаю, что библиотеки с питоническим API в меньшей степени нуждаются в этой системе типов, потому что ценность ее для них минимальна.

В этом сообщении Бенфилд привел доведенный до абсурда пример попытки определения типа для именованного аргумента `files` функции `requests.request()` (*https://docs.python-requests.org/en/master/api/#requests.request*):

```
Optional[
      Union[
               Mapping[
                        basestring,
                        Union[
                                 Tuple[basestring, Optional[Union[basestring, file]]],
                                 Tuple[basestring, Optional[Union[basestring, file]],
                                       Optional[basestring]],
                                 Tuple[basestring, Optional[Union[basestring, file]],
                                       Optional[basestring], Optional[Headers]]
                        ]
               ],
               Iterable[
                        Tuple[
                                 basestring,
                                 Union[
                                          Tuple[basestring, Optional[Union[basestring,
file]]],
                                          Tuple[basestring, Optional[Union[basestring,
file]],
                                                Optional[basestring]],
                                          Tuple[basestring, Optional[Union[basestring,
file]],
                                                Optional[basestring], Optional[Headers]]
                                 ]
                        ]
               ]
      ]
```

При этом еще предполагается следующее определение:

```
Headers = Union[
      Mapping[basestring, basestring],
      Iterable[Tuple[basestring, basestring]],
]
```

Как вы думаете, достигла бы библиотека `requests` такого успеха, если бы сопровождающие настаивали на 100-процентном покрытии аннотациями типов? SQLAlchemy – еще один важный пакет, который плохо уживается с аннотациями.

Своим успехом эти библиотеки обязаны тем, что на всю катушку используют динамическую природу Python.

У аннотаций типов есть достоинства, но есть и цена. Во-первых, нужно потратить усилия, чтобы понять, как работает система типов. Но это одноразовые затраты.

А есть еще и постоянные, которые придется нести всегда.

Настаивая на вставке аннотаций везде и всюду, мы теряем часть выразительности Python. Такие замечательные средства, как распаковка аргументов – например, `config(**settings)`, – выходят за пределы возможностей программ проверки типов.

Если вы хотите подвергнуть вызов вида `config(**settings)` проверке типов, то должны будете выписать каждый аргумент. Это наводит меня на воспоминания о Turbo Pascal, на котором я писал 35 лет назад.

Библиотеки, в которых используется метапрограммирование, трудно или даже невозможно анотировать. Конечно, метапрограммирование можно употребить во вред, но одновременно это вещь, благодаря которой со многими Python-пакетами так приятно работать.

Если в крупных компаниях использование аннотаций типов станет непререкаемым требованием, не допускающим исключений, то держу пари, что вскоре люди начнут использовать кодогенераторы, чтобы уменьшить объем стереотипного кода, – типичная практика в менее динамичных языках.

В некоторых проектах и контекстах аннотации типов просто не имеют смысла. И даже в тех контекстах, где в большинстве случаев смысл есть, бывают исключения. Любая разумная политика использования аннотаций типов должна допускать исключения.

Алан Кэй, лауреат премии Тьюринга, один из пионеров объектно-ориентированного программирования, писал:

> Некоторые относятся к системе типов с религиозным рвением. Мне как математику идея систем типов нравится, но никто еще не придумал систему с достаточно широким охватом[1].

Спасибо Гвидо за факультативную типизацию. Давайте будем использовать ее так, как она задумана, и не пытаться аннотировать все вокруг в строгом соответствии со стилем кодирования, напоминающим Java 1.5.

Пригодность утиной типизации

Утиная типизация отвечает моему складу ума, а статическая утиная типизация – удачный компромисс, допускающий статическую проверку типов, не жертвуя гибкостью, которая в некоторых системах номинальной типизации возможна лишь ценой сильного усложнения – если вообще возможна.

До выхода PEP 544 вся идея аннотаций типов казалась мне в высшей степени чуждой Python. Я был очень рад, когда увидел, как `typing.Protocol` вписался в Python. Это внесло некоторое равновесие.

[1] Источник: «Интервью с Аланом Кэем» (*https://queue.acm.org/detail.cfm?id=1039523*).

Обобщенные или конкретные?

С точки зрения Python, использование термина «обобщенный» в системе типов вывернуто наизнанку. Обычно, говоря «обобщенный» (generic), имеют в виду «применимый к целому классу или группе» либо «без торговой марки»[1].

Сравним `list` и `list[str]`. Первый список является обобщенным: он принимает любой объект. Второй список конкретный: он принимает только `str`.

Однако в Java этот термин имеет смысл. До выхода Java 1.5 все коллекции в Java (кроме магического `array`) были «конкретными»: они могли хранить только ссылки на `Object`, поэтому для использования элементов, извлекаемых из коллекции, приходилось приводить тип. В Java 1.5 коллекции получили параметры-типы и стали «обобщенными».

[1] В этом смысле вместо «обобщенный» в русском языке употребляется слово «дженерик». – *Прим. перев.*

Глава 9

Декораторы и замыкания

> Многие были недовольны выбором названия «декоратор» для этого средства. И главная причина – несогласованность с использованием термина в книге «Банды четырех»[1]. Название декоратор, пожалуй, в большей степени связано с употреблением в области разработки компиляторов – обход и аннотирование синтаксического дерева.
>
> – PEP 318 «Decorators for Functions and Methods»

Декораторы функций дают возможность «помечать» функции в исходном коде, тем или иным способом дополняя их поведение. Это мощное средство, но для овладения им нужно понимать, что такое замыкание – то, что мы получаем, когда функция захватывает переменную, определенную вне ее тела.

Одно из самых плохо понимаемых зарезервированных слов в Python – `nonlocal`, оно появилось в версии Python 3.0. Программист на Python может безбедно существовать, и не используя его, если будет строго придерживаться объектно-ориентированной диеты, основанной на классах. Но если вы захотите реализовать собственные декораторы функций, то должны досконально разбираться в замыканиях, а тогда потребность в слове `nonlocal` становится очевидной.

Помимо применения при реализации декораторов, замыкания важны также для эффективного асинхронного программирования без обратных вызовов и для кодирования в функциональном стиле там, где это имеет смысл.

Конечная цель этой главы – точно объяснить, как работают декораторы – от простейших регистрационных до более сложных параметризованных. Но прежде нам предстоит рассмотреть следующие вопросы:

- как интерпретатор Python разбирает синтаксис декораторов;
- как Python решает, является ли переменная локальной;
- зачем нужны замыкания и как они работают;
- какие проблемы решает ключевое слово `nonlocal`.

Заложив этот фундамент, мы сможем перейти непосредственно к декораторам:

- реализация корректно ведущего себя декоратора;
- мощные декораторы в стандартной библиотеке: `@cache`, `@lru_cache` и `@singledispatch`;
- реализация параметризованного декоратора.

---

[1] Вышедшая в1995 году книга «Design Patterns» за авторством так называемой «банды четырех» (Gamma et al., Addison-Wesley).

Что нового в этой главе

Кеширующий декоратор `functools.cache` – новое средство в Python 3.9 – проще традиционного `functools.lru_cache`, поэтому я представляю его первым. А второй, включая его упрощенную форму, появившуюся в Python 3.8, рассматривается в разделе «Использование lru_cache».

Раздел «Обобщенные функции с одиночной диспетчеризацией» был расширен, и теперь в нем используются аннотации типов – рекомендуемый способ использования `functools.singledispatch` начиная с Python 3.7.

Раздел «Параметризованные декораторы» теперь включает пример 9.27, в котором декоратор реализован в виде класса.

Я перенес главу 10 «Реализация паттернов проектирования с помощью полноправных функций» в конец части II, чтобы сделать последовательность изложения более логичной. Раздел «Паттерн Стратегия, дополненный декоратором» теперь находится в этой главе вместе с другими вариантами паттерна Стратегия, основанными на использовании вызываемых объектов.

Начнем с самых базовых понятий, относящихся к декораторам, а затем обратимся к остальным перечисленным выше темам.

Краткое введение в декораторы

Декоратор – это вызываемый объект, который принимает другую функцию в качестве аргумента (декорируемую функцию).

Декоратор может производить какие-то операции с функцией и возвращает либо ее саму, либо другую заменяющую ее функцию или вызываемый объект[1].

Иначе говоря, в предположении, что существует декоратор с именем `decorate`, следующий код:

```
@decorate
def target():
    print('running target()')
```

эквивалентен такому:

```
def target():
    print('running target()')

target = decorate(target)
```

Конечный результат одинаков: в конце обоих фрагментов имя `target` связано с функцией `target`, которую вернул вызов `decorate(target)`. Это может быть функция, которая изначально называлась `target`, или какая-то другая.

Чтобы убедиться, что декорируемая функция действительно заменена, рассмотрим сеанс оболочки в примере 9.1.

[1] Если в предыдущем предложении заменить слово «функция» словом «класс», то получится краткое описание того, что делает декоратор класса. Декораторы классов рассматриваются в главе 24.

Пример 9.1. Декоратор обычно заменяет одну функцию другой

```
>>> def deco(func):
...     def inner():
...         print('running inner()')
...     return inner ❶
...
>>> @deco
... def target(): ❷
...     print('running target()')
...
>>> target() ❸
running inner()
2
>>> target ❹
<function deco.<locals>.inner at 0x10063b598>
```

❶ `deco` возвращает свой внутренний объект-функцию `inner`.
❷ `target` декорирована `deco`.
❸ При вызове декорированной функции `target` на самом деле выполняется `inner`.
❹ Инспекция показывает, что `target` теперь ссылается на `inner`.

Строго говоря, декораторы – не более чем синтаксический сахар. Как мы видели, всегда можно просто вызвать декоратор как обычный вызываемый объект, передав ему функцию. Иногда это действительно удобно, особенно для *метапрограммирования* – изменения поведения программы в процессе ее выполнения.

Следующие три факта – главное, что нужно знать о декораторах:

- декоратор – это функция или другой вызываемый объект;
- декоратор может заменить декорируемую функцию другой;
- декораторы выполняются сразу после загрузки модуля.

Теперь сосредоточим внимание на последнем моменте.

Когда Python выполняет декораторы

Главное свойство декораторов – то, что они выполняются сразу после определения декорируемой функции. Обычно на *этапе импорта* (т. е. когда Python загружает модуль). Рассмотрим скрипт *registration.py* в примере 9.2.

Пример 9.2. Модуль registration.py

```
registry = [] ❶

def register(func): ❷
    print(f'running register({func})') ❸
    registry.append(func) ❹
    return func ❺

@register ❻
def f1():
    print('running f1()')

@register
```

```
def f2():
    print('running f2()')

def f3(): ❼
    print('running f3()')

def main(): ❽
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__ == '__main__':
    main() ❾
```

❶ В `registry` хранятся ссылки на функции, декорированные `@register`.
❷ `register` принимает функцию в качестве аргумента.
❸ Показать, какая функция декорируется, – для демонстрации.
❹ Включить `func` в `registry`.
❺ Вернуть `func`: мы должны вернуть функцию, в данном случае возвращается та же функция, что была передана на входе.
❻ `f1` и `f2` декорированы `@register`.
❼ `f3` не декорирована.
❽ `main` распечатывает `registry`, затем вызывает `f1()`, `f2()` и `f3()`.
❾ `main()` вызывается только тогда, когда `registration.py` запускается как скрипт.

Будучи запущена как скрипт, программа `registration.py` выводит следующие строки:

```
$ python3 registration.py
running register(<function f1 at 0x100631bf8>)
running register(<function f2 at 0x100631c80>)
running main()
registry -> [<function f1 at 0x100631bf8>, <function f2 at
0x100631c80>]
running f1()
running f2()
running f3()
```

Отметим, что `register` выполняется (дважды) до любой другой функции в модуле. При вызове `register` получает в качестве аргумента декорируемый объект-функцию, например `<function f1 at 0x100631bf8>`.

После загрузки модуля в `registry` оказываются ссылки на две декорированные функции: `f1` и `f2`. Они, как и функция `f3`, выполняются только при явном вызове из `main`.

Если `registration.py` импортируется (а не запускается как скрипт), то вывод выглядит так:

```
>>> import registration
running register(<function f1 at 0x10063b1e0>)
running register(<function f2 at 0x10063b268>)
```

Если сейчас заглянуть в `registry`, то мы увидим:

```
>>> registration.registry
[<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]
```

Основная цель примера 9.2 – подчеркнуть, что декораторы функций выполняются сразу после импорта модуля, но сами декорируемые функции – только в результате явного вызова. В этом проявляется различие между *этапом импорта* и *этапом выполнения* в Python.

Регистрационные декораторы

По сравнению с типичным применением декораторов в реальных программах, пример 9.2 необычен в двух отношениях.

- Функция-декоратор определена в том же модуле, что и декорируемые функции. Настоящий декоратор обычно определяется в одном модуле и применяется к функциям из других модулей.
- Декоратор `register` возвращает ту же функцию, что была передана в качестве аргумента. На практике декоратор обычно определяет внутреннюю функцию и возвращает именно ее.

Хотя декоратор `register` из примера 9.2 возвращает декорированную функцию без изменения, эта техника не бесполезна. Подобные декораторы используются во многих веб-каркасах, написанных на Python, с целью добавления функций в некий центральный реестр, например для отображения образцов URL на функции, генерирующие HTTP-ответы. Такие регистрационные декораторы могут изменять декорируемую функцию, но это необязательно.

Мы увидим, как применяется регистрационный декоратор, в разделе «Паттерн Стратегия, дополненный декоратором» главы 10.

Большинство декораторов все же изменяют декорируемую функцию. Обычно для этого определяется некая внутренняя функция, которая заменяет декорируемую. Код, в котором используются внутренние функции, неизбежно опирается на замыкания. Чтобы понять, что такое замыкания, нам придется отступить назад и тщательно разобраться с тем, как в Python работают области видимости переменных.

Правила видимости переменных

В примере 9.3 мы определяем и тестируем функцию, которая читает две переменные: локальную переменную `a`, определенную как параметр функции, и переменную `b`, которая внутри функции вообще не определена.

Пример 9.3. Функция, читающая локальную и глобальную переменные

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> f1(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f1
NameError: global name 'b' is not defined
```

Ошибка не должна вызывать удивления. Но если продолжить пример 9.3 и присвоить значение глобальной переменной `b`, а затем вызвать `f1`, то все заработает:

```
>>> b = 6
>>> f1(3)
3
6
```

А теперь рассмотрим пример, который, возможно, вас удивит.

Взгляните на функцию `f2` в примере 9.4. Первые две строчки в ней такие же, как в `f1` из примера 9.3, но затем мы присваиваем значение переменной `b`. Однако функция завершается с ошибкой на втором предложении `print`, до присваивания.

Пример 9.4. Переменная `b` локальна, потому что ей присваивается значение в теле функции

```
>>> b = 6
>>> def f2(a):
...     print(a)
...     print(b)
...     b = 9
...
>>> f2(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f2
UnboundLocalError: local variable 'b' referenced before assignment
```

Отметим, что число `3` все же напечатано, следовательно, предложение `print(a)` было выполнено. Но вот до `print(b)` дело так и не дошло. Впервые увидев этот пример, я очень удивился, так как думал, что `6` будет напечатано – ведь существует глобальная переменная `b`, а присваивание локальной `b` производится уже после `print(b)`.

Однако же, компилируя тело этой функции, Python решает, что `b` – локальная переменная, т. к. ей присваивается значение внутри функции. Сгенерированный байт-код отражает это решение и пытается выбрать `b` из локального контекста. Позже, во время вызова `f2(3)`, тело `f2` успешно находит и печатает локальную переменную `a`, но при попытке получить значение локальной переменной `b` обнаруживает, что `b` не связана.

Это не ошибка, а осознанный выбор: Python не заставляет нас объявлять переменные, но предполагает, что всякая переменная, которой присваивается значение в теле функции, локальна. Это гораздо лучше поведения JavaScript, который тоже не требует объявлять переменные, но если вы забудете объявить переменную локальной (с помощью зарезервированного слова `var`), то можете случайно затереть одноименную глобальную переменную.

Если нам нужно, чтобы интерпретатор считал переменную `b` глобальной, несмотря на присваивание внутри функции, то придется добавить объявление `global`:

```
>>> b = 6
>>> def f3(a):
...     global b
...     print(a)
...     print(b)
...     b = 9
...
>>> f3(3)
3
6
>>> b
9
```

В примерах выше имеется две области видимости:

Глобальная область видимости модуля

Образована именами, которым присвоены значения, не объявленные ни в одном блоке класса или функции.

Локальная область видимости функции f3

Образована именами, которым присвоены значения как параметрам или непосредственно в теле функции.

Существует еще одна область видимости, из которой могут происходить переменные. Она называется *нелокальной* и является фундаментальной для замыканий; мы познакомимся с ней чуть ниже.

После этого краткого знакомства с принципом работы областей видимости в Python мы можем приступить к замыканиям. А вниманию тех, кому интересно посмотреть, чем отличается байт-код функций из примеров 9.3 и 9.4, предлагается следующая врезка.

Сравнение байт-кода

Модуль `dis` позволяет без труда дизассемблировать байт-код функций Python. В примерах 9.5 и 9.6 показан байт-код функций `f1` и `f2` из примеров 9.3 и 9.4.

Пример 9.5. Дизассемблированная функция `f1` из примера 9.3

```
>>> from dis import dis
>>> dis(f1)
  2           0 LOAD_GLOBAL              0 (print) ❶
              3 LOAD_FAST                0 (a) ❷
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  3          10 LOAD_GLOBAL              0 (print)
             13 LOAD_GLOBAL              1 (b) ❸
             16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             19 POP_TOP
             20 LOAD_CONST               0 (None)
             23 RETURN_VALUE
```

❶ Загрузить глобальное имя `print`.

❷ Загрузить локальное имя `a`.

❸ Загрузить глобальное имя `b`.

А теперь сравните с байт-кодом функции `f2` в примере 9.6.

Пример 9.6. Дизассемблированная функция `f1` из примера 9.4

```
>>> dis(f2)
  2           0 LOAD_GLOBAL              0 (print)
              3 LOAD_FAST                0 (a)
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  3          10 LOAD_GLOBAL              0 (print)
             13 LOAD_FAST                1 (b) ❶
             16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             19 POP_TOP

  4          20 LOAD_CONST               1 (9)
             23 STORE_FAST               1 (b)
             26 LOAD_CONST               0 (None)
             29 RETURN_VALUE
```

❶ Загрузить *локальное* имя `b`. Как видим, компилятор считает `b` локальной переменной, даже если присваивание `b` встречается позже, поскольку природа переменной – локальная она или нет – не должна приводить к изменению тела функции.

Виртуальная машина CPython, которая исполняет байт-код, – это стековая машина, т. е. операции `LOAD` и `POP` относятся к стеку. Дальнейшее описание кодов операций Python выходит за рамки этой книги, но они документированы в разделе, посвященном модулю `dis`: «Дизассемблер байт-кода Python» (*http://docs.python.org/3/library/dis.html*).

Замыкания

В блогосфере замыкания иногда путают с анонимными функциями. Причина тому историческая: определение функций внутри функций кажется делом необычным до тех пор, пока мы не начинаем пользоваться анонимными функциями. А замыкания вступают в игру только при наличии вложенных функций. Поэтому многие изучают обе концепции одновременно.

На самом деле замыкание – это функция, назовем ее `f`, с расширенной областью видимости, которая охватывает переменные, на которые есть ссылки в теле `f`, но которые не являются ни глобальными, ни локальными переменными `f`. Такие переменные должны происходить из локальной области видимости внешней функции, объемлющей `f`.

Не имеет значения, является функция анонимной или нет; важно лишь, что она может обращаться к неглобальным переменным, определенным вне ее тела.

Эту идею довольно трудно переварить, поэтому лучше продемонстрировать ее на примере.

Рассмотрим функцию `avg`, которая вычисляет среднее продолжающегося ряда чисел, например среднюю цену закрытия биржевого товара за всю историю торгов. Каждый день ряд пополняется новой ценой, а при вычислении среднего учитываются все прежние цены.

Если начать с чистого листа, то функцию `avg` можно было бы использовать следующим образом:

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Откуда берется `avg` и где она хранит предыдущие значения?
Для начала покажем реализацию, основанную на классах.

Пример 9.7. average_oo.py: класс для вычисления накопительного среднего

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total / len(self.series)
```

Класс `Averager` создает вызываемые объекты:

```
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

А теперь покажем функциональную реализацию с использованием функции высшего порядка `make_averager`.

Пример 9.8. average.py: функция высшего порядка для вычисления накопительного среднего

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total / len(series)

    return averager
```

При обращении к `make_averager` возвращается объект-функция `averager`. При каждом вызове `averager` добавляет переданный аргумент в конец списка `series` и вычисляет текущее среднее, как показано в примере 9.9.

Пример 9.9. Тестирование функции из примера 9.8

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(15)
12.0
```

Обратите внимание на сходство обоих примеров: мы обращаемся к `Averager()` или к `make_averager()`, чтобы получить вызываемый объект `avg`, который обновляет временной ряд и вычисляет текущее среднее. В примере 9.7 `avg` – экземпляр `Averager`, а в примере 9.8 – внутренняя функция `averager`. И в том, и в другом случае мы просто вызываем `avg(n)`, чтобы добавить `n` в ряд и вычислить новое среднее.

Совершенно ясно, где хранит историю объект `avg` класса `Averager`: в атрибуте экземпляра `self.series`. Но где находит `series` функция `avg` из второго примера?

Обратите внимание, что `series` – локальная переменная `make_averager`, потому что инициализация `series = []` производится в теле этой функции. Но к моменту вызова `avg(10)` функция `make_averager` уже вернула управление, и ее локальная область видимости уничтожена.

Внутри `averager` `series` является *свободной переменной*. Этот технический термин означает, что переменная не связана в локальной области видимости. См. рис. 9.1.

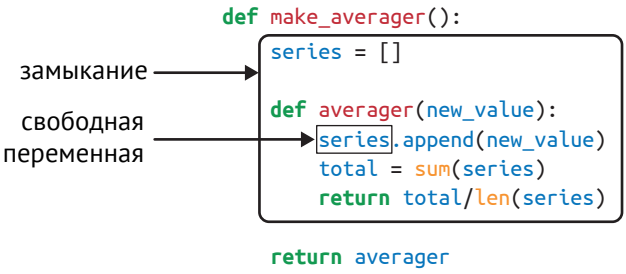


Рис. 9.1. Замыкание `averager` расширяет область видимости функции, включая в нее привязку свободной переменной `series`

Инспекция возвращенного объекта `averager` показывает, что Python хранит имена локальных и свободных переменных в атрибуте `__code__`, который представляет собой откомпилированное тело функции. Это показано в примере 9.10.

Пример 9.10. Инспекция функции, созданной функцией `make_averager` из примера 9.8

```
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
```

Значение переменной `series` хранится в атрибуте `__closure__` возвращенной функции `avg`. Каждому элементу `avg.__closure__` соответствует имя в `avg.__code__.co_freevars`. Эти элементы называются ячейками (`cells`), и у каждого из них есть

атрибут `cell_contents`, где можно найти само значение. Эти атрибуты демонстрируются в примере 9.11.

Пример 9.11. Продолжение примера 9.9

```
>>> avg.__code__.co_freevars
('series',)
>>> avg.__closure__
(<cell at 0x107a44f78: list object at 0x107a91a48>,)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
```

Резюмируем: замыкание – это функция, которая запоминает привязки свободных переменных, существовавшие на момент определения функции, так что их можно использовать впоследствии при вызове функции, когда область видимости, в которой она была определена, уже не существует.

Отметим, что единственная ситуация, когда функции может понадобиться доступ к внешним неглобальным переменным, – это когда она вложена в другую функцию и эти переменные являются частью локальной области видимости внешней функции.

Объявление nonlocal

Приведенная выше реализация функции `make_averager` неэффективна. В примере 9.8 мы храним все значения во временном ряде и вычисляем их сумму при каждом вызове `averager`. Лучше было бы хранить предыдущую сумму и количество элементов, тогда, зная эти два числа, можно вычислить новое среднее.

Реализация в примере 9.12 некорректна и приведена только в педагогических целях. Сможете ли вы найти ошибку?

Пример 9.12. Неправильная функция высшего порядка для вычисления накопительного среднего без хранения всей истории

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

При попытке выполнить этот код получится вот что:

```
>>> avg = make_averager()
>>> avg(10)
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'count' referenced before
assignment
>>>
```

Проблема в том, что предложение `count += 1` означает то же самое, что `count = count + 1`, где `count` – число или любой неизменяемый тип. То есть мы по сути дела присваиваем `count` значение в теле `averager`, делая ее тем самым локальной переменной. То же относится к переменной `total`.

В примере 9.8 этой проблемы не было, потому что мы ничего не присваивали переменной `series`; мы лишь вызывали `series.append` и передавали ее функциям `sum` и `len`. То есть воспользовались тем, что список – изменяемый тип.

Однако переменные неизменяемых типов – числа, строки, кортежи и т. д. – разрешается только читать, но не изменять. Если попытаться перепривязать такую переменную, как в случае `count = count + 1`, то мы неявно создадим локальную переменную `count`. Она уже не является свободной и потому не запоминается в замыкании.

Чтобы обойти эту проблему, в Python 3 было добавлено объявление `nonlocal`. Оно позволяет пометить переменную как свободную, даже если ей присваивается новое значение внутри функции. В таком случае изменяется привязка, хранящаяся в замыкании. Корректная реализация функции `make_averager` показана в примере 9.13.

Пример 9.13. Вычисление накопительного среднего без хранения всей истории (исправленный вариант с `nonlocal`)

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

Поняв, как используется `nonlocal`, суммируем все, что мы знаем о поиске переменных в Python.

Логика поиска переменных

Встретив определение функции, компилятор байт-кода Python определяет, как найти встречающуюся в ней переменную `x`, руководствуясь следующими правилами[1]:

- если имеется объявление `global x`, то `x` берется из него и присваивается глобальной переменной `x` уровня модуля[2];
- если имеется объявление `nonlocal x`, то `x` берется из него и присваивается локальной переменной `x` в ближайшей объемлющей функции, в которой `x` определена;
- если `x` – параметр или ей присвоено значение в теле функции, то `x` – локальная переменная;

[1] Благодарю технического рецензента Леонардо Рохаэля, который предложил эту сводку.

[2] В Python нет области видимости уровня программы, только уровня модуля.

- если на `x` имеется ссылка, но значение ей не присвоено и параметром она тоже не является, то:
  - `x` ищется в локальных областях видимости тел объемлющих функций (нелокальных областях видимости);
  - если она не найдена в объемлющих областях видимости, то читается из глобальной области видимости модуля;
  - если она не найдена и в глобальной области видимости, то читается из `__builtins__.__dict__`.

Теперь, познакомившись с замыканиями в Python, мы можем продемонстрировать эффективную реализацию декораторов с помощью вложенных функций.

Реализация простого декоратора

В примере 9.14 показан декоратор, который хронометрирует каждый вызов декорируемой функции и печатает затраченное время, переданные аргументы и результат.

Пример 9.14. Простой декоратор для вывода времени выполнения функции

```
import time

def clock(func):
    def clocked(*args): ❶
        t0 = time.perf_counter()
        result = func(*args) ❷
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
        return result ❸
    return clocked
```

❶ Определить внутреннюю функцию `clocked`, принимающую произвольное число позиционных аргументов.
❷ Эта функция работает только потому, что замыкание `clocked` включает свободную переменную `func`.
❸ Вернуть внутреннюю функцию взамен декорируемой.

В примере 9.15 демонстрируется использование декоратора `clock`.

Пример 9.15. Использование декоратора `clock`

```
import time
from clockdeco0 import clock

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)
```

```
if __name__ == '__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'Calling factorial(6)')
    print('6! =', factorial(6))
```

Вот что выводит этот код:

```
$ python3 clockdeco_demo.py
**************************************** Calling snooze(.123)
[0.12363791s] snooze(0.123) -> None
**************************************** Calling factorial(6)
[0.00000095s] factorial(1) -> 1
[0.00002408s] factorial(2) -> 2
[0.00003934s] factorial(3) -> 6
[0.00005221s] factorial(4) -> 24
[0.00006390s] factorial(5) -> 120
[0.00008297s] factorial(6) -> 720
6! = 720
```

Как это работает

Напомним, что код

```
@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)
```

на самом деле эквивалентен следующему:

```
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

factorial = clock(factorial)
```

То есть в обоих случаях декоратор `clock` получает функцию `factorial` в качестве аргумента `func` (см. пример 9.14). Затем он создает и возвращает функцию `clocked`, которую интерпретатор Python за кулисами связывает с именем `factorial`. На самом деле если импортировать модуль `clockdeco_demo` и вывести атрибут `__name__` функции `factorial`, то мы увидим:

```
>>> import clockdeco_demo
>>> clockdeco_demo.factorial.__name__
'clocked'
>>>
```

Таким образом, `factorial` действительно хранит ссылку на функцию `clocked`. Начиная с этого момента при каждом вызове `factorial(n)` выполняется `clocked(n)`. А делает `clocked` вот что:

1. Запоминает начальный момент времени `t0`.
2. Вызывает исходную функцию `factorial` и сохраняет результат.
3. Вычисляет, сколько прошло времени.
4. Форматирует и печатает собранные данные.
5. Возвращает результат, сохраненный на шаге 2.

Это типичное поведение декоратора: заменить декорируемую функцию новой, которая принимает те же самые аргументы и (как правило) возвращает то,

что должна была бы вернуть декорируемая функция, но при этом произвести какие-то дополнительные действия.

В книге Гамма и др. «Паттерны проектирования» краткое описание паттерна Декоратор начинается словами: «Динамически добавляет объекту новые обязанности». Декораторы функций отвечают этому описанию. Но на уровне реализации декораторы в Python имеют мало общего с классическим Декоратором, описанным в оригинальной книге. Ниже, во врезке «Поговорим», я еще вернусь к этой теме.

Декоратор `clock`, реализованный в примере 9.14, имеет ряд недостатков: он не поддерживает именованные аргументы и маскирует атрибуты `__name__` и `__doc__` декорированной функции. В примере 9.16 используется декоратор `functools.wraps`, который копирует необходимые атрибуты из `func` в `clocked`. К тому же в этой новой версии правильно обрабатываются именованные аргументы.

Пример 9.16. clockdeco.py: улучшенный декоратор `clock`

```
import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_lst = [repr(arg) for arg in args]
        arg_lst.extend(f'{k}={v!r}' for k, v in kwargs.items())
        arg_str = ', '.join(arg_lst)
        print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
        return result
    return clocked
```

Декоратор `functools.wraps` – лишь один из нескольких готовых декораторов в стандартной библиотеке. В следующем разделе мы рассмотрим самый впечатляющий декоратор в модуле `functools`: `cache`.

Декораторы в стандартной библиотеке

В Python есть три встроенные функции, предназначенные для декорирования методов: `property`, `classmethod` и `staticmethod`. Функцию `property` мы обсудим в разделе «Использование свойств для контроля атрибутов» главы 22, а остальные – в разделе «Декораторы classmethod и staticmethod» главы 11.

В примере 9.16 мы видели еще один важный декоратор – `functools.wraps`, вспомогательное средство для построения корректных декораторов. Некоторые из наиболее интересных декораторов в стандартной библиотеке – `cache`, `lru_cache` и `singledispatch` – определены в модуле `functools`. Их мы далее и рассмотрим.

Запоминание с помощью functools.cache

Декоратор `functools.lru_cache` реализует *запоминание*[1] (memoization): прием оптимизации, смысл которого заключается в сохранении результатов предыдущих дорогостоящих вызовов функции, что позволяет избежать повторного вычисления с теми же аргументами, что и раньше.

Декоратор `functools.cache` был добавлен в версии Python 3.9. Если вы хотите выполнить приведенные ниже примеры в Python 3.8, замените `@cache` на `@lru_cache`. В более ранних версиях Python нужно вызывать декоратор, записывая `@lru_cache()`, как объясняется в разделе «Использование `lru_cache`».

Продемонстрируем применение `cache` на примере медленной рекурсивной функции вычисления *n*-го числа Фибоначчи.

Пример 9.17. Очень накладный рекурсивный способ вычисления *n*-го числа Фибоначчи

```
from clockdeco import clock

@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)

if __name__ == '__main__':
    print(fibonacci(6))
```

Вот результат работы *fibo_demo.py*. Все строки, кроме последней, выведены декоратором `clock`:

```
$ python3 fibo_demo.py
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00007892s] fibonacci(2) -> 1
[0.00000095s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00003815s] fibonacci(2) -> 1
[0.00007391s] fibonacci(3) -> 2
[0.00018883s] fibonacci(4) -> 3
[0.00000000s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
[0.00000119s] fibonacci(1) -> 1
[0.00004911s] fibonacci(2) -> 1
[0.00009704s] fibonacci(3) -> 2
[0.00000000s] fibonacci(0) -> 0
[0.00000000s] fibonacci(1) -> 1
[0.00002694s] fibonacci(2) -> 1
[0.00000095s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
```

[1] Уточним: в слове *memoization* (*https://en.wikipedia.org/wiki/Memoization*) нет опечатки. В информатике этот термин хоть и отдаленно связан с «memorization» (сохранение в памяти), но означают они не одно и то же.

```
[0.00000095s] fibonacci(1) -> 1
[0.00005102s] fibonacci(2) -> 1
[0.00008917s] fibonacci(3) -> 2
[0.00015593s] fibonacci(4) -> 3
[0.00029993s] fibonacci(5) -> 5
[0.00052810s] fibonacci(6) -> 8
8
```

Непроизводительные затраты бросаются в глаза: `fibonacci(1)` вызывается восемь раз, `fibonacci(2)` – пять раз и т. д. Но если добавить две строчки, чтобы задействовать `cache`, то производительность резко возрастет.

Пример 9.18. Более быстрая реализация с использованием кеширования

```
import functools

from clockdeco import clock

@functools.cache ❶
@clock ❷
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)

if __name__ == '__main__':
    print(fibonacci(6))
```

❶ Эта строка работает в версии Python 3.9 и более поздних. Альтернативы для более ранних версий Python описаны в разделе «Использование `lru_cache`».

❷ Это пример композиции декораторов: `@cache()` применяется к функции, возвращенной декоратором `@clock`.

Композиция декораторов

Чтобы понять, в чем смысл композиции декораторов, вспомните, что `@` – синтаксический сахар для применения декорирующей функции к функции, которая находится под ней. Если декораторов несколько, то они ведут себя как вложенные вызовы функций. Запись

```
@alpha
@beta def my_fn():
    ...
```

семантически эквивалентна следующей:

```
my_fn = alpha(beta(my_fn))
```

Иными словами, сначала применяется декоратор `beta`, а возвращенная им функция передается декоратору `alpha`.

Благодаря использованию декоратора `cache` функция `fibonacci` вызывается всего один раз для каждого значения *n*:

```
$ python3 fibo_demo_lru.py
[0.00000119s] fibonacci(0) -> 0
[0.00000119s] fibonacci(1) -> 1
[0.00010800s] fibonacci(2) -> 1
[0.00000787s] fibonacci(3) -> 2
[0.00016093s] fibonacci(4) -> 3
[0.00001216s] fibonacci(5) -> 5
[0.00025296s] fibonacci(6) -> 8
```

В другом тесте для вычисления `fibonacci(30)` программа из примера 9.18 выполнила 31 вызов за 0,00017 с, тогда как программа без кеширования из примера 9.17 обращалась к функции `fibonacci(1)` 832 040 раз, а всего выполнида 2 692 537 вызовов и затратила на это 12,09 с на ноутбуке с процессором Intel Core i7.

Все аргументы, принимаемые декорированной функцией, должны быть *хешируемыми*, потому что `cache` пользуется словарем `dict` для хранения результатов, а его ключи образованы из позиционных и именованных аргументов, переданных при вызовах.

Но `@cache` умеет не только исправлять плохо написанные рекурсивные алгоритмы, во всем блеске он проявляется, когда нужно получать информацию от удаленных API.

`functools.cache` может занять всю имеющуюся память, если в кеше очень много элементов. Я считаю, что ее стоит использовать только в командных скриптах, работающих недолго. Для долгоживущих процессов я рекомендую декоратор `functools.lru_cache` с подходящим параметром `maxsize`, о котором речь пойдет в следующем разделе.

Использование lru_cache

Декоратор `functools.cache` на самом деле является простой оберткой вокруг появившейся раньше функции `functools.lru_cache`, более гибкой и совместимой с Python 3.8 и более ранними версиями.

Основное преимущество `@lru_cache` в том, что объем потребляемой памяти можно ограничить с помощью параметра `maxsize`, для которого по умолчанию подразумевается довольно консервативное значение 128, т. е. в любой момент времени в кеше может быть не более 128 элементов.

Акроним LRU расшифровывается как «Least Recently Used» (последний использованный); это означает, что элементы, к которым давно не было обращений, вытесняются, чтобы освободить место для новых.

Начиная с Python 3.8 `lru_cache` можно применять двумя способами. Вот как выглядит тот, что проще:

```
@lru_cache
def costly_function(a, b):
    ...
```

Другой способ существует еще со времен версии Python 3.2 и выглядит как вызов функции со скобками `()`:

```
@lru_cache()
def costly_function(a, b):
    ...
```

В обоих случаях используются параметры по умолчанию, а именно:

`maxsize=128`

Задает максимальное число элементов в кеше. После заполнения кеша элемент, который дольше других не использовался, вытесняется, и его место занимает новый элемент. Для достижения оптимальной производительности `maxsize` должен быть степенью 2. Если `maxsize=None`, то логика LRU отключается, поэтому кеш работает быстрее, но элементы никогда не вытесняются, что может привести к перерасходу памяти. Это именно то, что делает `@functools.cache`.

`typed=False`

Определяет, следует ли хранить элементы разного типа раздельно. Например, в конфигурации по умолчанию элементы типа `float` и `integer`, признанные равными, хранятся лишь один раз, т. е. вызовы `f(1)` и `f(1.0)` приведут к помещению в кеш только одного элемента. Если `typed=True`, то такие вызовы приведут к созданию двух разных элементов кеша.

Ниже приведен пример вызова `@lru_cache` со значениями параметров, отличающимися от умалчиваемых:

```
@lru_cache(maxsize=2**20, typed=True)
def costly_function(a, b):
    ...
```

А теперь изучим еще один мощный декоратор, `functools.singledispatch`.

Обобщенные функции с одиночной диспетчеризацией

Пусть требуется написать инструмент для отладки веб-приложений. Мы хотим, чтобы он умел генерировать HTML-представления объектов Python разного типа.

Можно было бы начать с такой функции:

```
import html

def htmlize(obj):
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'
```

Она будет работать для любого типа Python, но нам хотелось бы, чтобы для некоторых типов генерировались специальные представления. Вот несколько примеров.

`str`

Заменять внутренние символы новой строки строкой `'<br>\n'` и использовать теги `<p>` вместо `<pre>`.

`int`

Показывать число в десятичном и шестнадцатеричном виде (`bool` – специальный случай).

`list`

Выводить HTML-список, в котором каждый элемент отформатирован в соответствии со своим типом.

`float` и `Decimal`

Выводить значение, как обычно, а также в виде дроби (почему бы и нет?).

Желательное поведение показано в примере 9.19.

Пример 9.19. Функция `htmlize()` генерирует HTML-представление объектов разных типов

```
>>> htmlize({1, 2, 3}) ❶
'<pre>{1, 2, 3}</pre>'
>>> htmlize(abs)
'<pre>&lt;built-in function abs&gt;</pre>'
>>> htmlize('Heimlich & Co.\n- a game') ❷
'<p>Heimlich &amp; Co.<br/>\n- a game</p>'
>>> htmlize(42) ❸
'<pre>42 (0x2a)</pre>'
>>> print(htmlize(['alpha', 66, {3, 2, 1}])) ❹
<ul>
<li><p>alpha</p></li>
<li><pre>66 (0x42)</pre></li>
<li><pre>{1, 2, 3}</pre></li>
</ul>
>>> htmlize(True) ❺
'<pre>True</pre>'
>>> htmlize(fractions.Fraction(2, 3)) ❻
'<pre>2/3</pre>'
>>> htmlize(2/3) ❼
'<pre>0.6666666666666666 (2/3)</pre>'
>>> htmlize(decimal.Decimal('0.02380952'))
'<pre>0.02380952 (1/42)</pre>
```

❶ Оригинальная функция зарегистрирована для `object`, поэтому обрабатывает все типы аргументов, для которых не нашлось другой реализации.

❷ Объекты типа `str` также HTML-экранируются, но помещаются между тегами `<p>` и `</p>` и перед каждым `'.\n'` вставляется `<br/>`.

❸ Число типа `int` показывается в десятичном и шестнадцатеричном виде между тегами `<pre>` и `</pre>`.

❹ Каждый элемент списка форматируется в соответствии со своим типом, а вся последовательность оформляется как HTML-список.

❺ Хотя `bool` – подтип `int`, он обрабатывается специальным образом.

❻ Показывать `Fraction` как обыкновенную дробь.

❼ Показывать `float` и `Decimal` в виде приближенной обыкновенной дроби.

Функция singledispatch

Поскольку в Python нет механизма перегрузки методов, как в Java, мы не можем создать варианты `htmlize` с разными сигнатурами для каждого типа данных, который желательно обрабатывать специальным образом. Возможное решение состоит в том, чтобы преобразовать `htmlize` в функцию диспетчеризации, содержащую предложение `if` с несколькими ветвями `elif`, в каждой из которых вызывается некая специализированная функция: `htmlize_str`, `htmlize_int` и т. д. Но такое решение не поддается расширению пользователями модуля и слишком неуклюже: со временем диспетчер `htmlize` чрезмерно разрастется, а связь между ним и специализированными функциями станет недопустимо тесной.

Декоратор `functools.singledispatch` позволяет каждому модулю вносить свой вклад в общее решение, так что пользователь легко может добавить специализированную функцию, даже не имея возможности изменять класс. Обычная функция, декорированная `@singledispatch`, становится точкой входа для *обобщенной функции*: группы функций, выполняющих одну и ту же логическую операцию по-разному в зависимости от типа первого аргумента. Именно это и называется *одиночной диспетчеризацией*. Если бы для выбора конкретных функций использовалось больше аргументов, то мы имели бы множественную диспетчеризацию. В примере 9.20 показано, как это делается.

Декоратор `functools.singledispatch` существует, начиная с версии Python 3.4, но поддерживает аннотации типов, только начиная с версии Python 3.7. Последние две функции в примере 9.20 иллюстрируют синтаксис, который работает во всех версиях, начиная с Python 3.4.

Пример 9.20. Декоратор `singledispatch` создает функцию `@htmlize.register` для объединения нескольких функций в одну обобщенную

```
from functools import singledispatch
from collections import abc
import fractions
import decimal
import html
import numbers

@singledispatch ❶
def htmlize(obj: object) -> str:
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'

@htmlize.register ❷
def _(text: str) -> str: ❸
    content = html.escape(text).replace('\n', '<br/>\n')
    return f'<p>{content}</p>'

@htmlize.register ❹
def _(seq: abc.Sequence) -> str:
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'

@htmlize.register ❺
def _(n: numbers.Integral) -> str:
return f'<pre>{n} (0x{n:x})</pre>'

@htmlize.register ❻
def _(n: bool) -> str:
    return f'<pre>{n}</pre>'

@htmlize.register(fractions.Fraction) ❼
def _(x) -> str:
    frac = fractions.Fraction(x)
    return f'<pre>{frac.numerator}/{frac.denominator}</pre>'

@htmlize.register(decimal.Decimal) ❽
```

```
@htmlize.register(float)
def _(x) -> str:
    frac = fractions.Fraction(x).limit_denominator()
    return f'<pre>{x} ({frac.numerator}/{frac.denominator})</pre>'
```

❶ `@singledispatch` помечает базовую функцию, которая обрабатывает тип `object`.
❷ Каждая специализированная функция снабжается декоратором `@«base».register`.
❸ Тип первого аргумента, переданного во время выполнения, определяет, когда будет использоваться это конкретное определение функции. Имена специализированных функций несущественны, и это подчеркнуто выбором _ в качестве имени[1].
❹ Для каждого типа, нуждающегося в специальной обработке, регистрируется новая функция с подходящей аннотацией типа в первом параметре.
❺ Абстрактные базовые классы `numbers` полезны в сочетании с `singledispatch`[2].
❻ `bool` *является подтипом* `numbers.Integral`, но `singledispatch` ищет реализацию с самым специфичным подходящим типом независимо от порядка появления в программе.
❼ Если вы не хотите или не можете добавить аннотации типов в декорированную функцию, то можете передать тип декоратору `@«base».register`. Этот синтаксис работает начиная с версии Python 3.4.
❽ Декоратор `@«base».register` возвращает недекорированную функцию, поэтому можно компоновать их, чтобы зарегистрировать два или более типов для одной и той же реализации[3].

Если есть возможность, регистрируйте специализированные функции для обработки ABC (абстрактных классов), например `numbers.Integral` или `abc.MutableSequence`, а не конкретных реализаций, например `int` или `list`. Это позволит программе поддержать более широкий спектр совместимых типов. Например, расширение Python может предоставлять альтернативы типу `int` с фиксированной длиной в битах в качестве подклассов `numbers.Integral`[4].

Использование ABC или `typing.Protocol` в сочетании с `@singledispatch` позволит программе поддержать существующие или будущие классы, являющиеся настоящими или виртуальными подклассами этих ABC или реализующие указанный протокол. Применение ABC и концепция виртуального подкласса – темы главы 13.

---

[1] К сожалению, Муру 0.770 ругается, когда видит несколько функций с одинаковым именем.

[2] Несмотря на предупреждение в разделе «Падение числовой башни» главы 8, числовые ABC не объявлены нерекомендуемыми, и их можно найти в коде Python 3.

[3] Быть может, когда-нибудь мы сможем выразить это с помощью одного непараметризованного декоратора `@htmlize.register` и аннотации типа, в которой используется `Union`, но когда я попытался это сделать, Python возбудил исключение с сообщением о том, что `Union` – не класс. Так что хотя описанный в PEP 484 синтаксис поддерживается декоратором `@singledispatch`, с семантикой еще не все в порядке.

[4] Например, NumPy реализует несколько машинно-ориентированных типов целых и с плавающей точкой (*https://numpy.org/doc/stable/user/basics.types.html*).

Замечательное свойство механизма `singledispatch` состоит в том, что специализированные функции можно зарегистрировать в любом месте системы, в любом модуле. Если впоследствии вы добавите модуль, содержащий новый пользовательский тип, то сможете без труда написать новую специализированную функцию для обработки этого типа. А также реализовать функции обработки для классов, которые вы не писали и не можете изменить.

Декоратор `singledispatch` – продуманное дополнение к стандартной библиотеке, его возможности шире, чем описано выше. Документ PEP 443 «Single-dispatch generic functions» (*https://www.python.org/dev/peps/pep-0443/*) – хорошее справочное руководство, но в нем не упоминаются аннотации типов, добавленные позже. Документация по модулю `functools` доработана и более актуальна, в частности содержит несколько примеров в разделе, посвященном `singledispatch`.

Декоратор `@singledispatch` задуман не для того, чтобы перенести в Python перегрузку методов в духе Java. Один класс с несколькими перегруженными вариантами метода лучше одной функции с длинной цепочкой предложений `if/elif/elif/elif`. Но оба решения грешат тем, что поручают слишком много обязанностей одной единице программы – классу или функции. Преимущество `@singledispath` – в поддержке модульного расширения: каждый модуль может зарегистрировать специализированную функцию для того типа, который поддерживает. На практике вы вряд ли стали бы помещать все реализации обобщенных функций в один модуль, как в примере 9.20.

Мы видели несколько декораторов, принимающих аргументы, например `@lru_cache()` и `htmlize.register(float)` в примере 9.20. В следующем разделе описано, как создавать декораторы с параметрами.

Параметризованные декораторы

Разбирая декоратор, встретившийся в исходном коде, Python берет декорируемую функцию и передает ее в качестве первого аргумента функции-декоратору. А как сделать, чтобы декоратор принимал и другие аргументы? Ответ таков: написать фабрику декораторов, которая принимает эти аргументы и возвращает декоратор, который затем применяется к декорируемой функции. Непонятно? Естественно. Начнем с примера, основанного на простейшем из рассмотренных до сих пор декораторов: `register` (см. пример 9.21).

Пример 9.21. Сокращенный модуль *registration.py* из примера 9.2, повторен для удобства

```
registry = []

def register(func):
    print(f'running register({func})')
    registry.append(func)
    return func

@register
def f1():
    print('running f1()')

print('running main()')
```

```
print('registry ->', registry)
f1()
```

Параметризованный регистрационный декоратор

Чтобы функцию регистрации, вызываемую декоратором `register`, можно было активировать и деактивировать, мы снабдим ее необязательным параметром `active`: если он равен `False`, то декорируемая функция не регистрируется. В примере 9.22 показано, как это делается. Концептуально новая функция `register` – не декоратор, а фабрика декораторов. Будучи вызвана, она возвращает настоящий декоратор, который применяется к декорируемой функции.

Пример 9.22. Чтобы новый декоратор `register` мог принимать параметры, его следует вызывать как функцию

```
registry = set() ❶

def register(active=True):
    def decorate(func): ❷
        print('running register' ❸
              f'(active={active})->decorate({func})')
        if active: ❹
            registry.add(func)
        else:
            registry.discard(func) ❺

        return func ❻
    return decorate ❼

@register(active=False) ❽
def f1():
    print('running f1()')

@register() ❾
def f2():
    print('running f2()')

def f3():
    print('running f3()')
```

❶ Теперь `registry` имеет тип `set`, чтобы ускорить добавление и удаление функций.
❷ Функция `register` принимает необязательный именованный аргумент.
❸ Собственно декоратором является внутренняя функция `decorate`, она принимает в качестве аргумента функцию.
❹ Регистрируем `func`, только если аргумент `active` (определенный в замыкании) равен `True`.
❺ Если не `active` и функция `func` присутствует в `registry`, удаляем ее.
❻ Поскольку `decorate` – декоратор, он должен возвращать функцию.
❼ Функция `register` – наша фабрика декораторов, поэтому она возвращает `decorate`.
❽ Фабрику `@register` следует вызывать как функцию, передавая ей нужные параметры.
❾ Даже если параметров нет, `register` все равно нужно вызывать как функцию – `@register()`, – чтобы она вернула настоящий декоратор `decorate`.

Идея в том, что функция `register()` возвращает декоратор `decorate`, который затем применяется к декорируемой функции.

Код из примера 9.22 находится в модуле *registration_param.py*. Если его импортировать, получится вот что:

```
>>> import registration_param
running register(active=False)->decorate(<function f1 at 0x10063c1e0>)
running register(active=True)->decorate(<function f2 at 0x10063c268>)
>>> registration_param.registry
[<function f2 at 0x10063c268>]
```

Заметим, что в `registry` присутствует только функция `f2`, а функция `f1` туда не попала, т. к. фабрике декораторов `register` был передан аргумент `active=False`, поэтому метод `decorate`, примененный к `f1`, не добавил ее в `registry`.

Если бы мы использовали `register` как обычную функцию без символа `@`, то для декорирования функции `f`, т. е. для добавления ее в `registry`, нужно было бы написать `register()(f)`, а чтобы не добавлять `f` в реестр (или удалить оттуда) – `register(active=False)(f)`. В примере 9.23 показано, как добавлять функции в реестр `registry` и удалять из него.

Пример 9.23. Использование модуля `registration_param` из примера 9.22

```
running register(active=False)->decorate(<function f1 at 0x10073c1e0>)
running register(active=True)->decorate(<function f2 at 0x10073c268>)
>>> registry ❶
{<function f2 at 0x10073c268>}
>>> register()(f3) ❷
running register(active=True)->decorate(<function f3 at 0x10073c158>)
<function f3 at 0x10073c158>
>>> registry ❸
{<function f3 at 0x10073c158>, <function f2 at 0x10073c268>}
>>> register(active=False)(f2) ❹
running register(active=False)->decorate(<function f2 at 0x10073c268>)
<function f2 at 0x10073c268>
>>> registry ❺
{<function f3 at 0x10073c158>}
```

❶ После импортирования модуля `f2` оказывается в `registry`.
❷ Выражение `register()` возвращает декоратор `decorate`, который затем применяется к `f3`.
❸ В предыдущей строке функция `f3` была добавлена в `registry`.
❹ Этот вызов удаляет `f2` из `registry`.
❺ Убедиться, что `f3` осталась в `registry`.

Механизм работы параметризованных декораторов довольно сложен; рассмотренный выше пример проще, чем в большинстве случаев. Параметризованные декораторы обычно заменяют декорируемую функцию, а в их конструкторах необходим еще один уровень вложенности. В экскурсию по такой пирамиде функций мы отправимся в следующем разделе.

Параметризованный декоратор clock

В этом разделе мы вернемся к декоратору `clock` и добавим возможность передавать ему строку, управляющую форматом вывода. См. пример 9.24.

Для простоты код в примере 9.24 основан на первоначальной реализации `clock` в примере 9.14, а не на улучшенной реализации из примера 9.16, в которой использовался декоратор `@functools.wraps`, добавляющий еще один слой.

Пример 9.24. Модуль `clockdeco_param.py`: параметризованный декоратор `clock`

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

def clock(fmt=DEFAULT_FMT): ❶
    def decorate(func): ❷
        def clocked(*_args): ❸
            t0 = time.perf_counter()
            _result = func(*_args) ❹
            elapsed = time.perf_counter() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args) ❺
            result = repr(_result) ❻
            print(fmt.format(**locals())) ❼
            return _result ❽
        return clocked ❾
    return decorate ❿

if __name__ == '__main__':

    @clock() ⓫
    def snooze(seconds):
        time.sleep(seconds)

    for i in range(3):
        snooze(.123)
```

❶ Теперь `clock` – наша фабрика параметризованных декораторов.
❷ `decorate` – это собственно декоратор.
❸ `clocked` обертывает декорированную функцию.
❹ `_result` – результат, возвращенный декорированной функцией.
❺ В `_args` хранятся фактические аргументы `clocked`, тогда как `args` – отображаемая строка.
❻ `result` – строковое представление `_result`, предназначенное для отображения.
❼ Использование `**locals()` позволяет ссылаться в `fmt` на любую локальную переменную `clocked`[1].

[1] Технический рецензент Мирослав Седивы заметил: «Это также означает, что линтеры будут сообщать о неиспользуемых переменных, потому что они, как правило, игнорируют `locals()`». Да, это еще один пример того, как инструменты статического анализа отваживают от использования тех динамических средств, из-за которых Python так привлекателен для меня и еще множества программистов. Чтобы удовлетворить линтер, я мог бы записать каждую локальную переменную дважды при вызове: `fmt.format(elapsed=elapsed, name=name, args=args, result=result)`. Но не буду. Если вы пользуетесь инструментами статического анализа, то важно знать, когда их следует игнорировать.

❽ `clocked` заменяет декорированную функцию, поэтому должна возвращать то, что вернула бы эта функция в отсутствие декоратора.
❾ `decorate` возвращает `clocked`.
❿ `clock` возвращает `decorate`.
⓫ В этом тесте `clock()` вызывается без аргументов, поэтому декоратор будет использовать форматную строку по умолчанию.

При выполнении программы из примера 9.24 печатается следующее:

```
$ python3 clockdeco_param.py
[0.12412500s] snooze(0.123) -> None
[0.12411904s] snooze(0.123) -> None
[0.12410498s] snooze(0.123) -> None
```

Для демонстрации новой функциональности в примерах 9.25 и 9.26 показаны еще два модуля, в которых используется `clockdeco_param`, а также результаты их выполнения.

Пример 9.25. clockdeco_param_demo1.py

```
import time
from clockdeco_param import clock

@clock('{name}: {elapsed}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Результат выполнения примера 9.25:

```
$ python3 clockdeco_param_demo1.py
snooze: 0.12414693832397461s
snooze: 0.1241159439086914s
snooze: 0.12412118911743164s
```

Пример 9.26. clockdeco_param_demo2.py

```
import time
from clockdeco_param import clock

@clock('{name}({args}) dt={elapsed:0.3f}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Результат выполнения примера 9.26:

```
$ python3 clockdeco_param_demo2.py
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
```

Леннарт Регебро – один из рецензентов первого издания этой книги – считает, что декораторы лучше писать как классы, реализующие метод `__call__`, а не как функции (как в примерах из этой главы). Согласен, что для нетривиальных декораторов такой подход разумнее. Но функции проще, когда требуется объяснить основную идею этого механизма. См. раздел «Дополнительная литература», а в особенности блог Грэхема Дамплтона и модуль `wrapt`, если хотите узнать, как пишутся реальные декораторы.

В следующем разделе показан пример декоратора, написанного в стиле, рекомендуемом Регебро и Дамплтоном.

Декоратор clock на основе класса

Напоследок в примере 9.27 приведена реализация параметризованного декоратора `clock` в виде класса с методом `__call__`. Сравните примеры 9.24 и 9.27. Какой вам больше нравится?

Пример 9.27. Модуль clockdeco_cls.py: параметризованный декоратор `clock`, реализованный в виде класса

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

class clock: ❶

    def __init__(self, fmt=DEFAULT_FMT): ❷
        self.fmt = fmt

    def __call__(self, func): ❸
        def clocked(*_args):
            t0 = time.perf_counter()
            _result = func(*_args) ❹
            elapsed = time.perf_counter() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args)
            result = repr(_result)
            print(self.fmt.format(**locals()))
            return _result
        return clocked
```

❶ Теперь фабрикой параметризованных декораторов является не внешняя функция `clock`, а класс `clock`. Я решил писать его имя со строчной буквы `c`, чтобы было понятно, что эту реализацию можно подставить вместо приведенной в примере 9.24.

❷ Аргумент, переданный `clock(my_format)`, присваивается параметру `fmt`. Конструктор класса возвращает экземпляр `clock`, а `my_format` сохраняется в `self.fmt`.

❸ Наличие метода `__call__` делает `clock` вызываемым объектом. При вызове этот объект заменяет декорированную функцию на `clocked`.

❹ `clocked` обертывает декорированную функцию.

На этом мы завершаем изучение декораторов функций. С декораторами классов мы познакомимся в главе 24.

Резюме

В этой главе мы рассматривали трудный материал, но я старался сделать путешествие по возможности комфортабельным, хотя дорога была ухабистой. Ведь мы, по существу, вступили на территорию метапрограммирования.

Мы начали с простого декоратора `@register` без внутренней функции и закончили параметризованным декоратором `@clock()` с двумя уровнями вложенных функций.

Регистрационные декораторы, хотя и простые по существу, находят реальные применения в развитых каркасах на Python. Мы воспользуемся идеей регистрации в одной из реализаций паттерна проектирования Стратегия в главе 10.

Для понимания механизма работы декораторов понадобилось разобраться в различиях между *этапом импорта* и *этапом выполнения*, в областях действия переменных, в замыканиях и в новом объявлении `nonlocal`. Свободное владение замыканиями и объявлением `nonlocal` важно не только при написании декораторов, но и при разработке событийно-ориентированных программ с графическим интерфейсом, для асинхронного ввода-вывода с обратными вызовами, а также для применения функционального стиля программирования, когда это имеет смысл.

Параметризованные декораторы почти всегда содержат по меньшей мере две вложенные функции, а иногда и больше, если мы хотим использовать `@functools.wraps` для создания декоратора, который лучше поддерживает некоторые продвинутые возможности. Одну такую возможность – композицию декораторов – мы рассмотрели в примере 9.18. Для более сложных декораторов реализация в виде класса может оказаться проще для чтения и сопровождения.

В качестве примеров параметризованных декораторов в стандартной библиотеке мы рассмотрели два впечатляющих декоратора из модуля `functools`: `@cache` и `@singledispatch`.

Дополнительная литература

В рецепте 26 из книги Brett Slatkin «Effective Python» *https://effectivepython.com/*, 2-е издание (Addison-Wesley), рассматриваются передовые практики написания декораторов функций и всегда рекомендуется использовать `functools.wraps`. Мы видели это в примере 9.16[1].

Грэхем Дамплтон опубликовал в своем блоге (*https://github.com/GrahamDumpleton/wrapt/blob/develop/blog/README.md*) серию статей о способах реализации корректно работающих декораторов, и первая из них называется «How You Implemented Your Python Decorator is Wrong» (Ваш способ реализации декоратора в Python неправильный) (*https://github.com/GrahamDumpleton/wrapt/blob/develop/blog/01-how-you-implemented-your-python-decorator-is-wrong.md*). Его обширный опыт в этой области аккуратно инкапсулирован в модуль `wrapt` (*http://wrapt.readthedocs.org/en/latest/*), написанный с целью упростить реализацию декораторов и динамических функций-оберток, которые поддерживают

[1] Я хотел сделать код как можно более простым, поэтому не во всех примерах следовал прекрасному совету Слаткина.

интроспекцию и корректно ведут себя, если еще раз подвергаются декорированию, а также в случае применения к методам и использования в качестве дескрипторов. Дескрипторы – тема главы 23.

В главе 9 «Метапрограммирование» книги David Beazley, Brian K. Jones «*Python Cookbook*», 3-е издание (O'Reilly), есть несколько рецептов – от элементарных декораторов до очень сложных, в том числе такого, который можно вызывать либо как обычный декоратор, либо как фабрику декораторов, например `@clock` или `@clock()`. Это рецепт 9.6 «Определение декоратора, принимающего необязательный аргумент».

Мишель Симионато написал пакет, имеющий целью «облегчить среднему программисту использование декораторов и популяризировать декораторы путем демонстрации различных нетривиальных примеров». На сайте PyPI пакет доступен под названием `decorator` (*https://pypi.python.org/pypi/decorator*).

Вики-страница Python Decorator Library (*https://wiki.python.org/moin/PythonDecoratorLibrary*), созданная, когда декораторы только появились в Python, содержит десятки примеров. Поскольку странице уже много лет, некоторые приемы устарели, но она по-прежнему остается источником новых идей.

В коротенькой статье «Closures in Python» (*http://effbot.org/zone/closure.htm*) в блоге Фредрика Лундха объясняется терминология замыканий.

В документе PEP 3104 «Access to Names in Outer Scopes» (*http://www.python.org/dev/peps/pep-3104/*) доступно описано объявление `nonlocal`, позволяющее перепривязывать имена, не являющиеся ни локальными, ни глобальными. Здесь же имеется отличный обзор подходов к этой задаче в других динамических языках (Perl, Ruby, JavaScript и т. д.), а также обсуждение плюсов и минусов различных проектных решений, возможных в Python.

Документ PEP 227 «Statically Nested Scopes» (*http://www.python.org/dev/peps/pep-0227/*) более теоретического характера содержит введение в механизм лексических областей видимости, который появился как факультативное средство в Python 2.1 и стал стандартным в Python 2.2. Здесь же дается обоснование и варианты реализации замыканий в Python.

В документе PEP 443 (*http://www.python.org/dev/peps/pep-0443/*) приводится обоснование и детальное описание создания обобщенных функций с помощью одиночной диспетчеризации. В старой (март 2005 года) статье в блоге Гвидо ван Россума «Five-Minute Multimethods in Python» (Мультиметоды в Python за пять минут) (*http://www.artima.com/weblogs/viewpost.jsp?thread=101605*) подробно рассматривается реализация обобщенных функций (или мультиметодов) с помощью декораторов. Код Гвидо поддерживает множественную диспетчеризацию (т. е. диспетчеризацию на основе нескольких позиционных аргументов). Этот код интересен прежде всего с педагогической точки зрения. Современная готовая к работе реализация обобщенных функций с множественной диспетчеризацией имеется в библиотеке Reg (*http://reg.readthedocs.org/en/latest/*) Мартина Фаассена, автора моделеориентированного и поддерживающего REST веб-каркаса Morepath (*http://morepath.readthedocs.org/en/latest/*).

Поговорим

Сравнение динамической и лексической областей действия

Проектировщик любого языка с полноправными функциями сталкивается со следующей проблемой: будучи полноправными объектами, функции определены в некоторой области видимости, но могут вызываться из других областей видимости. Вопрос: как вычислять свободные переменные? Самое простое, что сразу приходит в голову: «динамическая область видимости». Это означает, что при вычислении свободных переменных просматривается окружение, в котором функция вызывается.

Если бы в Python были динамические области видимости, но не было замыканий, то функцию `avg` – аналогичную той, что приведена в примере 9.8, – можно было бы написать так:

```
>>> ### это не настоящий сеанс оболочки Python! ###
>>> avg = make_averager()
>>> series = []❶
>>> avg(10)
10.0
>>> avg(11) ❷
10.5
>>> avg(12)
11.0
>>> series = [1] ❸
>>> avg(5)
3.0
```

❶ Перед тем как использовать `avg`, мы должны сами определить список `series = []`, поскольку `averager` (внутри `make_averager`) ссылается на список по этому имени.

❷ За кулисами `series` используется для хранения усредняемых значений.

❸ При выполнении присваивания `series = [1]` предыдущий список затирается. Это может произойти случайно, если одновременно вычисляются два независимых средних.

Функции должны быть черными ящиками, их реализация должна быть скрыта от пользователя. Но если в функции имеются динамические переменные, то при использовании динамических областей видимости программист обязан знать внутреннее устройство функции, чтобы правильно настроить ее окружение. После многих лет борьбы с языком подготовки документов LaTeX я наткнулся на великолепную книгу George Grätzer «Practical LaTeX» (Springer), из которой узнал, что для переменных LaTeX используется динамическая область видимости. Вот почему они вызывали у меня такие трудности!

В Emacs Lisp также используется динамическая область видимости, по крайней мере по умолчанию. Краткое пояснение см. в разделе «Динамическое связывание» (*https://www.gnu.org/software/emacs/manual/html_node/elisp/Dynamic-Binding.html*) руководства по Emacs Lisp.

Динамическую область видимости проще реализовать, и, наверное, именно поэтому Джон Маккарти выбрал такой путь при создании Lisp, первого языка, в котором появились полноправные функции. Статья Пола Грэхема «The

Roots of Lisp» (*http://www.paulgraham.com/rootsoflisp.html*) содержит доступное объяснение оригинальной статьи Маккарти о языке Lisp «Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I» (Рекурсивные функции символических выражений и их вычисление машиной, часть I) (*http://bit.ly/mccarthy_recursive*). Работа Маккарти – такой же шедевр, как Девятая симфония Бетховена. Пол Грэхем перевел ее для всех нас – с языка математики на английский, а затем на язык кода.

Из комментария Пола Грэхема также видно, что динамические области видимости далеко не тривиальны. Приведем цитату из его статьи:

> Красноречивым свидетельством того, какими опасностями чреваты динамические области видимости, является тот факт, что даже самый первый пример функции высшего порядка в Lisp не работал – именно из-за них. Быть может, в 1960 году Маккарти не вполне сознавал последствия использования динамических областей видимости. Как бы то ни было, они оставались в реализациях Lisp на удивление долго – пока Сассмен и Стил не разработали язык Scheme в 1975 году. Лексические области видимости не слишком усложняют определение eval, но могут затруднить написание компиляторов.

Сегодня лексическая область видимости считается нормой: свободные переменные вычисляются в том окружении, в котором функция определена. Лексические области видимости усложняют реализацию языков с полноправными функциями, потому что зависят от поддержки замыканий. С другой стороны, исходный код с лексическими областями видимости проще читать. В большинстве языков, придуманных после Algol, имеются лексические области видимости. Заметное исключение – JavaScript, в котором специальная переменная `this` вызывает недоразумения, потому что может иметь как лексическую, так и динамическую область видимости в зависимости от того, как написан код (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this*).

В течение многих лет лямбда-выражения в Python не поддерживали замыкания, что снискало им дурную славу среди адептов функционального программирования в блогосфере. Это было исправлено в версии Python 2.2 (декабрь 2001), но у блогосферы долгая память. С тех пор к конструкции `lambda` есть только одна претензия: синтаксические ограничения.

Декораторы в Python и паттерн проектирования Декоратор

Декораторы функций в Python согласуются с общим описанием паттерна Декоратор в книге Гамма и др. «Паттерны проектирования»: «Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности». На уровне реализации декораторы в Python не имеют ничего общего с классическим паттерном Декоратор, но какую-то аналогию провести можно.

В паттерне проектирования `Decorator` и `Component` – абстрактные классы. Экземпляр конкретного декоратора обертывает экземпляр конкретного компонента, чтобы расширить его поведение. Приведем цитату из «Паттернов проектирования»:

> Декоратор следует интерфейсу декорируемого объекта, поэтому его присутствие прозрачно для клиентов компонента. Декоратор переадресует запросы внутреннему компоненту, но может выполнять и дополнительные действия (например, рисовать рамку) до или после переадресации. Поскольку декораторы прозрачны, они могут вкладываться друг в друга, добавляя тем самым любое число новых обязанностей.

В Python декоратор играет роль конкретного подкласса `Decorator`, а внутренняя функция, которую он возвращает, является экземпляром декоратора. Возвращенная функция обертывает декорируемую функцию, которая может быть уподоблена компоненту в паттерне проектирования. Возвращенная функция прозрачна, потому что согласуется с интерфейсом компонента, ведь она принимает те же самые аргументы. Она переадресует вызов компоненту и может выполнять дополнительные действия до или после переадресации. Мы можем переформулировать последнее предложение из приведенной цитаты следующим образом: «Поскольку декораторы прозрачны, они могут вкладываться друг в друга, добавляя тем самым любое число новых видов поведения». Именно это свойство открывает возможность композиции декораторов.

Я вовсе не предлагаю использовать декораторы для реализации паттерна Декоратор в программах на Python. Хотя в некоторых специфических ситуациях это возможно, в общем случае паттерн Декоратор лучше реализовать с помощью классов, представляющих сам Декоратор и обертываемые им компоненты.

Глава 10

Реализация паттернов проектирования с помощью полноправных функций

> Соответствием паттернам качество не измеряется.
>
> – Ральф Джонсон, один из авторов классической книги «Паттерны проектирования»[1]

В программной инженерии *паттерном проектирования* (*https://en.wikipedia.org/wiki/Software_design_pattern*) называется общий рецепт решения типичной задачи проектирования. Для чтения этой главы знакомство с паттернами проектирования необязательно. Я буду объяснять паттерны, встречающиеся в примерах.

Применение паттернов проектирования было популяризировано в знаменательной книге Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software» (Addison-Wesley), авторы которой получили шутливое прозвище «Банда четырех». Книга представляет собой каталог 23 паттернов и описывает структурную организацию классов с примерами на C++, хотя предполагается, что они будут полезны и в других объектно-ориентированных языках.

Хотя паттерны проектирования от языка не зависят, это не значит, что любой паттерн применим к любому языку. Например, в главе 17 мы покажем, что эмулировать рецепт паттерна Итератор в Python не имеет смысла, потому что он встроен в сам язык и готов к использованию в форме генераторов, которым классы вообще не нужны, а кода требуется меньше, чем в классическом рецепте.

Авторы книги «Паттерны проектирования» признают во введении, что применимость паттернов зависит от реализации языка:

> Выбор языка программирования важен, поскольку он определяет точку зрения. В наших паттернах подразумевается использование возможностей Smalltalk и C++, и от этого выбора зависит, что реализовать легко, а что – трудно. Если бы мы имели в виду процедурные языки, то включили бы паттерны «Наследование», «Инкапсуляция» и «Полиморфизм». Неко-

[1] Со слайда к докладу «Root Cause Analysis of Some Faults in Design Patterns», прочитанному Ральфом Джонсоном на IME/CCSL, университет Сан-Паулу, 15 ноября 2014.

> торые из наших паттернов напрямую поддерживаются менее распространенными языками. Так, в языке CLOS есть мультиметоды, которые делают ненужным паттерн «Посетитель»[1].

В презентации 1996 года «Design Patterns in Dynamic Languages» (*http://norvig.com/design-patterns/*) Петер Норвиг утверждает, что 16 из 23 паттернов, описанных в оригинальной книге «Паттерны проектирования», в динамических языках «либо не видны, либо более просты» (слайд 9). Он говорил о языках Lisp и Dylan, но аналогичные динамические средства существуют и в Python. В частности, в контексте языков с полноправными функциями Норвиг предлагает переосмыслить паттерны Стратегия, Команда, Шаблонный метод и Посетитель.

Цель этой главы – показать, что в некоторых случаях функции могут проделать ту же работу, что и классы, а код при этом получается проще и короче. Мы переработаем паттерн Стратегия с помощью объектов-функций, убрав много стереотипного кода, и обсудим аналогичный подход к упрощению паттерна Команда.

Что нового в этой главе

Я перенес эту главу в конец части III, чтобы можно было применить регистрационный декоратор, описанный в разделе «Паттерн Стратегия, дополненный декоратором», а также использовать в примерах аннотации типов. Большинство встречающихся в этой главе аннотаций несложны, но код с ними читать проще.

Практический пример: переработка паттерна Стратегия

Стратегия – прекрасный пример паттерна проектирования, который в Python можно упростить путем использования функций как полноправных объектов. В следующем разделе мы опишем и реализуем Стратегию, сохраняя верность «классической» структуре, описанной в «Паттернах проектирования». Если вы знакомы с классическим паттерном, то можете сразу перейти к разделу «Функционально-ориентированная Стратегия», где код будет переработан, в результате чего его объем существенно уменьшится.

Классическая Стратегия

На UML-диаграмме классов (рис. 10.1) изображены взаимоотношения классов, участвующих в паттерне Стратегия.

В книге «Паттерны проектирования» паттерн Стратегия описывается следующим образом:

> Определить семейство алгоритмов, инкапсулировать каждый из них и сделать их взаимозаменяемыми. Стратегия позволяет заменять алгоритм независимо от использующих его клиентов.

[1] *Эрих Гамма*, *Ричард Хелм*, *Ральф Джонсон*, *Джон Влиссидес*. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Питер, 2001.

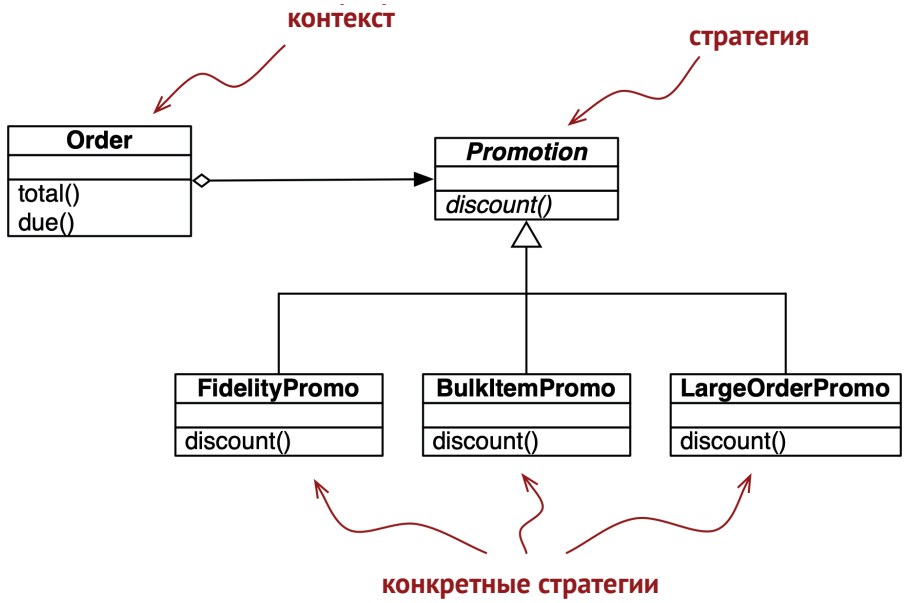


Рис. 10.1. UML-диаграмма классов для обработки скидок по заказам, реализованных в соответствии с паттерном Стратегия

Наглядный пример применения паттерна Стратегия к коммерческой задаче – вычисление скидок на заказы в соответствии с характеристиками заказчика или результатами анализа заказанных позиций.

Рассмотрим интернет-магазин со следующими правилами формирования скидок:

- заказчику, имеющему не менее 1000 баллов лояльности, предоставляется глобальная скидка 5 % на весь заказ;
- на позиции, заказанные в количестве не менее 20 единиц в одном заказе, предоставляется скидка 10 %;
- на заказы, содержащие не менее 10 различных позиций, предоставляется глобальная скидка 7 %.

Для простоты предположим, что к каждому заказу может быть применена только одна скидка.

UML-диаграмма классов для паттерна Стратегия показана на рис. 10.1. Ее участниками являются:

Контекст

Предоставляет службу, делегируя часть вычислений взаимозаменяемым компонентам, реализующим различные алгоритмы. В примере интернет-магазина контекстом является класс `Order`, который конфигурируется для применения поощрительной скидки по одному из нескольких алгоритмов.

Стратегия

Интерфейс, общий для всех компонентов, реализующих различные алгоритмы. В нашем примере эту роль играет абстрактный класс `Promotion`.

Конкретная стратегия

Один из конкретных подклассов Стратегии. В нашем случае реализованы три конкретные стратегии: `FidelityPromo`, `BulkPromo` и `LargeOrderPromo`.

Код в примере 10.1 следует изображенной на рис. 10.1 схеме. Как описано в «Паттернах проектирования», конкретная стратегия выбирается клиентом класса контекста. В нашем примере система, перед тем как создать объект заказа, должна каким-то образом выбрать стратегию предоставления скидки и передать ее конструктору класса `Order`. Вопрос о выборе стратегии не является предметом данного паттерна.

Пример 10.1. Реализация класса Order с помощью взаимозаменяемых стратегий предоставления скидки

```
from abc import ABC, abstractmethod
from collections.abc import Sequence
from decimal import Decimal
from typing import NamedTuple, Optional

class Customer(NamedTuple):
    name: str
    fidelity: int

class LineItem(NamedTuple):
    product: str
    quantity: int
    price: Decimal

    def total(self) -> Decimal:
        return self.price * self.quantity

class Order(NamedTuple): # контекст
    customer: Customer
    cart: Sequence[LineItem]
    promotion: Optional['Promotion'] = None

    def total(self) -> Decimal:
        totals = (item.total() for item in self.cart)
        return sum(totals, start=Decimal(0))

    def due(self) -> Decimal:
        if self.promotion is None:
            discount = Decimal(0)
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'

class Promotion(ABC): # Стратегия: абстрактный базовый класс
    @abstractmethod
    def discount(self, order: Order) -> Decimal:
        """Вернуть скидку в виде положительной суммы в долларах"""
```

```
class FidelityPromo(Promotion): # первая конкретная стратегия
    """5%-ная скидка для заказчиков, имеющих не менее 1000 баллов лояльности"""

    def discount(self, order: Order) -> Decimal:
        rate = Decimal('0.05')
        if order.customer.fidelity >= 1000:
            return order.total() * rate
        return Decimal(0)

class BulkItemPromo(Promotion): # вторая конкретная стратегия
    """10%-ная скидка для каждой позиции LineItem, в которой заказано
       не менее 20 единиц"""

    def discount(self, order: Order) -> Decimal:
        discount = Decimal(0)
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * Decimal('0.1')
            return discount

class LargeOrderPromo(Promotion): # третья конкретная стратегия
    """7%-ная скидка для заказов, включающих не менее 10 различных позиций"""

    def discount(self, order: Order) -> Decimal:
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * Decimal('0.07')
        return Decimal(0)
```

Отметим, что в примере 10.1 я сделал `Promotion` абстрактным базовым классом (ABC), чтобы можно было использовать декоратор `@abstractmethod` и тем самым прояснить структуру паттерна.

В примере 10.2 показаны тесты, которые проверяют работу модуля, реализующего описанные выше правила.

Пример 10.2. Пример использования класса `Order` с различными стратегиями скидок

```
>>> joe = Customer('John Doe', 0) ❶
>>> ann = Customer('Ann Smith', 1100)
>>> cart = (LineItem('banana', 4, Decimal('.5')), ❷
...         LineItem('apple', 10, Decimal('1.5')),
...         LineItem('watermelon', 5, Decimal(5)))
>>> Order(joe, cart, FidelityPromo()) ❸
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, FidelityPromo()) ❹
<Order total: 42.00 due: 39.90>
>>> banana_cart = (LineItem('banana', 30, Decimal('.5')), ❺
...                LineItem('apple', 10, Decimal('1.5')))
>>> Order(joe, banana_cart, BulkItemPromo()) ❻
<Order total: 30.00 due: 28.50>
>>> long_cart = tuple(LineItem(str(sku), 1, Decimal(1)) ❼
...                   for sku in range(10))
>>> Order(joe, long_cart, LargeOrderPromo()) ❽
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, LargeOrderPromo())
<Order total: 42.00 due: 42.00>
```

❶ Два заказчика: у `joe` 0 баллов лояльности, у `ann` – 1100.
❷ Одна корзина покупок с тремя позициями.
❸ Класс `FidelityPromo` не дает `joe` никаких скидок.
❹ `ann` получает скидку 5 %, поскольку имеет не менее 1000 баллов лояльности.
❺ В корзине `banana_cart` находится 30 бананов и 10 яблок.
❻ Класс `BulkItemPromo` дает `joe` скидку $1.50 на бананы.
❼ В заказе `long_order` имеется 10 различных позиций стоимостью $1.00 каждая.
❽ `joe` получает скидку 7 % на весь заказ благодаря классу `LargerOrderPromo`.

Пример 10.1 работает без нареканий, но ту же функциональность можно реализовать в Python гораздо короче, воспользовавшись функциями как объектами.

Функционально-ориентированная стратегия

Каждая конкретная стратегия в примере 10.1 – это класс с одним методом `discount`. К тому же объекты стратегии не имеют состояния (атрибутов экземпляра). Мы могли бы сказать, что они сильно напоминают функции, и были бы правы. В примере 10.3 код из примера 10.1 переработан – конкретные стратегии заменены простыми функциями, а абстрактный класс `Promo` исключен вовсе. В класс `Order` пришлось внести совсем немного изменений[1].

Пример 10.3. Класс `Order`, в котором стратегии предоставления скидок реализованы в виде функций

```
from collections.abc import Sequence
from dataclasses import dataclass
from decimal import Decimal
from typing import Optional, Callable, NamedTuple

class Customer(NamedTuple):
    name: str
    fidelity: int

class LineItem(NamedTuple):
    product: str
    quantity: int
    price: Decimal

    def total(self):
        return self.price * self.quantity

@dataclass(frozen=True)
class Order: # контекст
```

[1] Я был вынужден переделать класс `Order`, добавив декоратор `@dataclass` из-за ошибки в Муру. Можете не обращать внимания на эту деталь, потому что класс работает и с `NamedTuple`, как в примере 10.1. Если `Order` является `NamedTuple`, то Муру 0.910 «падает» при проверке аннотации типа для `promotion`. Я пытался добавить `# type ignore` в эту конкретную строку, но Муру все равно падает. Ту же самую аннотацию типа Муру обрабатывает правильно, если `Order` снабжен декоратором `@dataclass`. Проблема #9397 остается нерешенной по состоянию на 19 июля 2021 года. Надеюсь, что она будет решена к моменту, когда вы будете это читать.

```
    customer: Customer
    cart: Sequence[LineItem]
    promotion: Optional[Callable[['Order'], Decimal]] = None ❶

    def total(self) -> Decimal:
        totals = (item.total() for item in self.cart)
        return sum(totals, start=Decimal(0))

    def due(self) -> Decimal:
        if self.promotion is None:
            discount = Decimal(0)
        else:
            discount = self.promotion(self) ❷
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'

❸

def fidelity_promo(order: Order) -> Decimal: ❹
    """5%-ная скидка для заказчиков, имеющих не менее 1000 баллов лояльности"""
    if order.customer.fidelity >= 1000:
        return order.total() * Decimal('0.05')
    return Decimal(0)

def bulk_item_promo(order: Order) -> Decimal:
    """10%-ная скидка для каждой позиции LineItem, в которой заказано
       не менее 20 единиц»»»
    discount = Decimal(0)
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * Decimal('0.1')
        return discount

def large_order_promo(order: Order) -> Decimal:
    """7%-ная скидка для заказов, включающих не менее 10 различных позиций"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * Decimal('0.07')
    return Decimal(0)
```

❶ Эта аннотация типов означает, что `promotion` может быть равно `None` или вызываемому объекту, который принимает аргумент типа `Order` и возвращает `Decimal`.
❷ Для вычисления скидки просто вызываем функцию `self.promotion()`.
❸ Абстрактного класса больше нет.
❹ Каждая стратегия является функцией.

Почему self.promotion(self)?

В классе `Order` переменная `promotion` не является методом. Это атрибут класса, по стечению обстоятельств оказавшийся вызываемым объектом. Чтобы его вызвать, мы должны предоставить экземпляр `Order` – в данном случае `self`. Поэтому `self` и встречается в этом выражении дважды.

> В разделе «Методы являются дескрипторами» главы 23 мы объясним механизм автоматического связывания методов с экземплярами. Но к `promotion` он неприменим, потому что это не метод.

Код в примере 10.3 короче, чем в примере 10.1. Пользоваться новым классом `Order` также несколько проще, как показано в тестах ниже.

Пример 10.4. Пример использования класса `Order`, в котором стратегии скидки реализованы в виде функций

```
>>> joe = Customer('John Doe', 0) ❶
>>> ann = Customer('Ann Smith', 1100)
>>> cart = [LineItem('banana', 4, Decimal('.5')),
... LineItem('apple', 10, Decimal('1.5')),
... LineItem('watermelon', 5, Decimal(5))]
>>> Order(joe, cart, fidelity_promo) ❷
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, fidelity_promo)
<Order total: 42.00 due: 39.90>
>>> banana_cart = [LineItem('banana', 30, Decimal('.5')),
... LineItem('apple', 10, Decimal('1.5'))]
>>> Order(joe, banana_cart, bulk_item_promo) ❸
<Order total: 30.00 due: 28.50>
>>> long_cart = [LineItem(str(item_code), 1, Decimal(1))
... for item_code in range(10)]
>>> Order(joe, long_cart, large_order_promo)
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, large_order_promo)
<Order total: 42.00 due: 42.00>
```

❶ Те же тестовые фикстуры, что в примере 10.1.
❷ Для применения стратегии скидки к объекту `Order` нужно просто передать функцию скидки в качестве аргумента.
❸ Здесь и в следующем тесте используются разные функции скидки.

По поводу выносок в примере 10.4: нет необходимости создавать новый объект скидки для каждого заказа – функции и так готовы к применению.

Интересно, что авторы «Паттернов проектирования» замечают: «в большинстве случаев объекты-стратегии подходят как приспособленцы»[1]. В другой части книги паттерн Приспособленец определяется так: «Приспособленец – это разделяемый объект, который можно использовать одновременно в нескольких контекстах»[2]. Разделение рекомендуется для того, чтобы сэкономить на стоимости создания экземпляров конкретных стратегий, которые многократно применяются в каждом новом контексте – в нашем примере к каждому объекту `Order`. Поэтому в целях преодоления недостатка паттерна Стратегия – высоких накладных расходов во время выполнения – авторы рекомендуют применять еще один паттерн. И тем самым увеличиваются объем и сложность сопровождения кода.

В более сложном случае, когда у конкретных стратегий имеется внутреннее состояние, может оказаться необходимым как-то комбинировать части паттернов Стратегия и Приспособленец. Но часто у конкретных стратегий нет

[1] «Паттерны проектирования», стр. 309.

[2] Там же, стр. 192.

внутреннего состояния, они имеют дело только с данными из контекста. И тогда ничто не мешает использовать обычные функции вместо написания классов с единственным методом, которые реализуют интерфейс, объявленный еще в одном классе. Функция обходится дешевле экземпляра пользовательского класса, и отпадает надобность в паттерне Приспособленец, потому что каждая функция-стратегия создается только один раз – когда Python компилирует модуль. Обычная функция как раз и является «разделяемым объектом, который можно использовать одновременно в нескольких контекстах».

Теперь, когда мы знаем, как реализовать паттерн Стратегия, перед нами открываются и другие возможности. Допустим, мы хотим создать «метастратегию», которая выбирает наилучшую скидку для данного объекта `Order`. В следующих разделах мы продолжим переработку и покажем различные подходы к реализации этого требования, используя функции и модули как объекты.

Выбор наилучшей стратегии: простой подход

Используя тех же заказчиков и корзины покупок, что в примере 10.4, мы добавим еще три теста.

Пример 10.5. Функция `best_promo` применяет все стратегии и возвращает наибольшую скидку

```
>>> Order(joe, long_cart, best_promo) ❶
<Order total: 10.00 due: 9.30>
>>> Order(joe, banana_cart, best_promo) ❷
<Order total: 30.00 due: 28.50>
>>> Order(ann, cart, best_promo) ❸
<Order total: 42.00 due: 39.90>
```

❶ Для покупателя `joe` функция `best_promo` выбрала стратегию `larger_order_promo`.
❷ Здесь `joe` получил скидку от `bulk_item_promo` за заказ большого числа бананов.
❸ Несмотря на очень простую корзину, `best_promo` дала лояльному покупателю `ann` скидку согласно стратегии `fidelity_promo`.

Реализация `best_promo` очень проста и показана в примере 10.6.

Пример 10.6. `best_promo` находит максимальную скидку, перебирая все функции

```
promos = [fidelity_promo, bulk_item_promo, large_order_promo] ❶

def best_promo(order: Order) -> Decimal: ❷
    """Вычислить наибольшую скидку"""
    return max(promo(order) for promo in promos) ❸
```

❶ `promos`: список стратегий, реализованных в виде функций.
❷ `best_promo` получает объект `Order` в качестве аргумента, как и другие функции `*_promo`.
❸ С помощью генераторного выражения мы применяем к `order` каждую функцию из списка `promos` и возвращаем максимальную вычисленную скидку.

Код в примере 10.6 работает бесхитростно: `promos` – это список функций. Сжившись с идеей о том, что функции – полноправные объекты, вы будете воспринимать и структуры, содержащие функции, как нечто естественное.

Пример 10.6 работает, и читать код легко, но все же в нем есть некоторое дублирование, которое может приводить к тонкой ошибке: чтобы добавить новую стратегию скидки, нужно написать функцию и не забыть добавить ее в список `promos`, иначе новая стратегия будет работать, если явно передать ее в качестве аргумента `Order`, но `best_promotion` ее рассматривать не будет.

Ниже описано два решения этой проблемы. Читайте дальше.

Поиск стратегий в модуле

Модули в Python также являются полноправными объектами, и в стандартной библиотеке есть несколько функций для работы с ними. В документации встроенная функция `globals` описана следующим образом:

`globals()`

> Возвращает словарь, представляющий текущую таблицу глобальных символов. Это всегда словарь текущего модуля (внутри функции или метода это тот модуль, где данная функция или метод определены, а не модуль, из которого они вызваны).

В примере 10.7 показан не вполне честный способ использования `globals`, позволяющий `best_promo` автоматически находить все доступные функции `*_promo`.

Пример 10.7. Список `promos` строится путем просмотра глобального пространства имен модуля

```
from decimal import Decimal
from strategy import Order
from strategy import (
    fidelity_promo, bulk_item_promo, large_order_promo ❶
)
promos = [promo for name, promo in globals().items() ❷
          if name.endswith('_promo') and ❸
             name != 'best_promo' ❹
]

def best_promo(order: Order) -> Decimal: ❺
    """Вычислить наибольшую скидку"""
    return max(promo(order) for promo i n promos)
```

❶ Импортировать функции скидок, чтобы они были доступны в глобальном пространстве имен[1].
❷ Перебрать все имена в словаре, возвращенном функцией `globals()`.
❸ Оставить только имена с суффиксом `_promo`.
❹ Отфильтровать саму функцию `best_promo`, чтобы не было бесконечной рекурсии.
❺ Сама функция `best_promo` не изменилась.

Другой способ собрать вместе все стратегии скидки – создать отдельный модуль и поместить в него все функции-стратегии, кроме `best_promo`.

[1] flake8 и VS Code ругаются, что эти имена импортированы, но не используются. По определению, инструменты статического анализа не способны понять динамическую структуру Python. Если тупо следовать каждому их совету, то мы начнем писать угрюмый и многословный код а-ля Java с синтаксисом Python.

Пример 10.8 отличается только тем, что список функций-стратегий строится путем просмотра специального модуля `promotions`. Отметим, что для работы этого кода необходимо импортировать модуль `promotions`, а также модуль `inspect`, в котором находятся высокоуровневые функции интроспекции.

Пример 10.8. Список `promos` строится путем интроспекции нового модуля `promotions`

```
from decimal import Decimal
import inspect

from strategy import Order
import promotions

promos = [func for _, func in inspect.getmembers(promotions,
inspect.isfunction)]

def best_promo(order: Order) -> Decimal:
    """Вычислить наибольшую скидку"""
    return max(promo(order) for promo in promos)
```

Функция `inspect.getmembers` возвращает атрибуты объекта – в данном случае модуля `promotions` – возможно, отфильтрованные предикатом (булевой функцией). Мы пользуемся предикатом `inspect.isfunction`, чтобы получить только имеющиеся в модуле функции.

Код в примере 10.8 работает независимо от имен функций, важно лишь, чтобы модуль `promotions` содержал только функции вычисления скидки для переданного заказа. Конечно, это некое неявное предположение: если кто-нибудь включит в модуль `promotions` функцию с другой сигнатурой, то при попытке применить ее к заказу функция `best_promo` завершится с ошибкой.

Можно было бы отбирать функции более строго, например анализируя их аргументы. Но цель примера 10.8 – не предложить полное решение, а показать один из возможных путей использования интроспекции модуля.

Есть и более явный подход к динамическому отбору функций вычисления скидки – воспользоваться декоратором. Этому посвящен следующий раздел.

Паттерн Стратегия, дополненный декоратором

Напомним, что в примере 10.6 мы столкнулись с проблемой повторения имен функций в определениях и в списке `promos`, который используется функцией `best_promo` для вычисления максимально возможной скидки. Такое повторение плохо тем, что программист может добавить новую функцию-стратегию, забыв включить ее в список `promos`, и тогда `best_promo` молча проигнорирует новую стратегию, а в системе появится тонкая ошибка. В примере 10.9 эта проблема решается с помощью регистрационного декоратора.

Пример 10.9. Список `promos` заполняется декоратором `promotion`

```
Promotion = Callable[[Order], Decimal]

promos: list[Promotion] = [] ❷

def promotion(promo: Promotion) -> Promotion: ❷
```

```
    promos.append(promo)
    return promo

def best_promo(order: Order) -> Decimal:
    """Вычислить наибольшую скидку"""
    return max(promo(order) for promo in promos) ❸

@promotion ❹
def fidelity(order: Order) -> Decimal:
    """5%-ная скидка для заказчиков, имеющих не менее 1000 баллов лояльности"""
    if order.customer.fidelity >= 1000:
        return order.total() * Decimal('0.05')
    return Decimal(0)

@promotion
def bulk_item(order: Order) -> Decimal:
    """10%-ная скидка для каждой позиции LineItem, в которой заказано
       не менее 20 единиц»»»
    discount = Decimal(0)
    for item in order.cart:
    if item.quantity >= 20:
        discount += item.total() * Decimal('0.1')
    return discount

@promotion
def large_order(order: Order) -> Decimal:
    """7%-ная скидка для заказов, включающих не менее 10 различных позиций"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * Decimal('0.07')
    return Decimal(0)
```

❶ Список `promos` глобален на уровне модуля и вначале пуст.

❷ Регистрационный декоратор `promotion` возвращает функцию `promo_func` без изменения, но добавляет ее в список `promos`.

❸ Функция `best_promo` не изменяется, поскольку зависит только от списка `promos`.

❹ Все функции, декорированные `@promotion`, добавлены в `promos`.

По сравнению с другими решениями, представленными выше, у этого есть несколько преимуществ.

- Функции, реализующие стратегии вычисления скидки, не обязаны иметь специальные имена – суффикс `_promo` не нужен.
- Декоратор `@promotion` ясно описывает назначение декорируемой функции и без труда позволяет временно отменить предоставление ссылки: достаточно закомментировать декоратор.
- Стратегии скидки можно определить в других модулях, в любом месте системы; главное – чтобы к ним применялся декоратор `@promotion`.

В следующем разделе мы обсудим паттерн Команда, который часто реализуют с помощью классов с единственным методом, хотя достаточно и обычной функции.

Паттерн Команда

Команда – еще один паттерн проектирования, который можно упростить с помощью передачи функций в качестве аргументов. На рис. 10.2 показана диаграмма классов для этого паттерна.

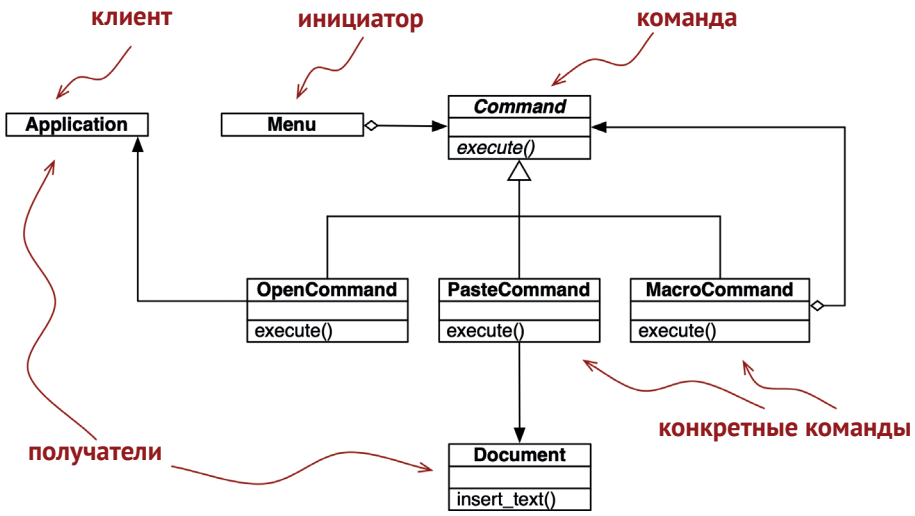


Рис. 10.2. UML-диаграмма классов для управляемого меню текстового редактора, реализованного с применением паттерна Команда. У каждой команды может быть свой получатель: объект, выполняющий действие. Для команды `PasteCommand` получателем является `Document`, а для `OpenCommand` – приложение

Цель Команды – разорвать связь между объектом, инициировавшим операцию (Инициатором), и объектом, который ее реализует (Получателем). В примере из «Паттернов проектирования» инициаторами являются пункты меню в графическом редакторе, а получателями – редактируемый документ или само приложение.

Идея в том, чтобы поместить между инициатором и получателем объект `Command`, который реализует интерфейс с единственным методом `execute`, вызывающим какой-то метод Получателя для выполнения желаемой операции. Таким образом, Инициатор ничего не знает об интерфейсе Получателя, так что, написав подклассы `Command`, можно адаптировать различных получателей. Инициатор конфигурируется конкретной командой и вызывает ее метод `execute`. Отметим, что на рис. 10.2 показан, в частности, класс `MacroCommand`, который может хранить последовательность команд; его метод `execute()` вызывает одноименный метод каждой хранимой команды.

Авторы «Паттернов проектирования» пишут: «Команды – объектно-ориентированная замена обратным вызовам». Вопрос: а нужна ли нам объектно-ориентированная замена обратным вызовам? Иногда да, а иногда и нет.

Вместо того чтобы передавать Инициатору объект `Command`, мы можем передать ему обычную функцию. И вызывать Инициатор будет не метод `command.execute()`, а просто функцию `command()`. Класс `MacroCommand` можно реализовать с по-

мощью класса, в котором реализован специальный метод `__call__`. Тогда экземпляры `MacroCommand` будут вызываемыми объектами, содержащими список функций для последующего вызова (см. пример 10.10).

Пример 10.10. В каждом объекте `MacroCommand` хранится внутренний список команд

```
class MacroCommand:
    """Команда, выполняющая список команд"""

    def __init__(self, commands):
        self.commands = list(commands) ❶

    def __call__(self):
        for command in self.commands: ❷
            command()
```

❶ Построение списка, инициализированного аргументом `commands`, гарантирует, что это итерируемый объект, и сохраняет локальную копию ссылок на команды в каждом экземпляре `MacroCommand`.

❷ При вызове экземпляра `MacroCommand` последовательно вызываются все команды из списка `self.commands`.

Для менее тривиальных применений паттерна Команда – например, для поддержки операции отмены – простой функции обратного вызова может не хватить. Но даже в этом случае Python предлагает две альтернативы, заслуживающие внимания.

- Вызываемый объект наподобие `MacroCommand` из примера 10.10 может хранить произвольное состояние и предоставлять другие методы в дополнение к `__call__`.
- Для запоминания внутреннего состояния функции между ее вызовами можно воспользоваться замыканием.

На этом мы завершаем переосмысление паттерна Команда, навеянное применением полноправных функций. На верхнем уровне этот подход близок к использованному в паттерне Стратегия: заменить вызываемыми объектами экземпляры класса-участника, реализующего интерфейс с единственным методом. Ведь любой вызываемый объект в Python и так реализует интерфейс с единственным методом, а именно методом `__call__`.

Резюме

Как отметил Петер Норвиг спустя два года после выхода классической книги «Паттерны проектирования»: «16 из 23 паттернов в языках Lisp и Dylan имеют существенно более простую реализацию, чем в C++, по крайней мере в некоторых ситуациях» (слайд 9 из презентации Норвига «Паттерны проектирования в динамических языках» (*http://www.norvig.com/design-patterns/index.htm*)). Python обладает некоторыми динамическими средствами, имеющимися в языках Lisp и Dylan, в частности полноправными функциями, которым в основном и посвящена эта часть книги.

В том же выступлении на праздновании 20-й годовщины выхода «Паттернов проектирования», цитата из которого послужила эпиграфом к этой главе,

Ральф Джонсон говорил, что одной из неудач книги стало «чрезмерно большое внимание к паттернам как конечным точкам, а не шагам проектирования»[1]. В этой главе мы взяли в качестве отправной точки паттерн Стратегия и показали, как упростить его работоспособную реализацию путем использования полноправных функций.

Во многих случаях функции или вызываемые объекты оказываются более естественным способом реализации обратных вызовов в Python, чем рабское следование описаниям паттернов Стратегия или Команда, приведенным в книге «Паттерны проектирования». Переработка Стратегии и обсуждение Команды – примеры более общей ситуации: если встречается паттерн или API, который нуждается в компоненте с единственным методом, и этот метод имеет такое общее название, как «execute», «run» или «doIt», то такой паттерн или API часто проще реализовать с помощью полноправных функций или иных вызываемых объектов. При этом объем стереотипного кода уменьшается.

Дополнительная литература

В рецепте 8.21 «Реализация паттерна Посетитель» из книги David Beazley, Brian K. Jones «*Python Cookbook*», 3-е издание (O'Reilly), предложена элегантная реализация паттерна Посетитель, в которой класс `NodeVisitor` обращается с методами, как с полноправными объектами.

По паттернам проектирования выбор литературы для программиста на Python не так широк, как для других языков.

Книга Gennadiy Zlobin «Learning Python Design Patterns» (Packt) – единственная целиком посвященная паттернам в Python. Но она очень короткая (100 страниц), и в ней рассмотрены только восемь из 23 оригинальных паттернов.

Книга Tarek Ziade «Expert Python Programming» (Packt) – одна из лучших на рынке для программистов на Python среднего уровня, а в последней главе «Useful Design Patterns» описаны семь классических паттернов с точки зрения Python.

Алекс Мартелли несколько раз выступал с докладами на тему паттернов проектирования в Python. В сети опубликована видеозапись его презентации на конференции EuroPython 2011 (*https://pyvideo.org/europython-2011/python-design-patterns.html*), а на его личном сайте также выложен набор слайдов (*http://www.aleax.it/gdd_pydp.pdf*). В разные годы мне встречались наборы слайдов разной комплектности и видео разной продолжительности, так что имеет смысл поискать повнимательнее, указав в запросе имя автора и слова «Python Design Patterns». В издательстве мне сказали, что Мартелли пишет книгу на эту тему. Я обязательно куплю ее, когда она выйдет.

Существует много книг по паттернам проектирования в контексте Java, из них я хотел бы выделить книгу Eric Freeman, Bert Bates, Kathy Sierra, Elisabeth Robson «Head First Design Patterns», 2-е издание (O'Reilly). В ней объясняются 16 из 23 классических паттернов. Если вам нравится неформальный стиль серии «Head First» и требуется введение в эту тему, то это как раз то, что надо. Книга ориентирована на Java, но во втором издании отражено добавление

[1] Из доклада «Root Cause Analysis of Some Faults in Design Patterns», прочитанного Джонсоном на IME-USP 15 ноября 2014 года.

полноправных функций в Java, так что некоторые примеры стали ближе к коду, который мы пишем на Python.

Тем, кого интересует свежий взгляд на паттерны с точки зрения динамического языка с динамической типизацией и полноправными функциями, стоит прочитать книгу Russ Olsen «Design Patterns in Ruby» (Addison-Wesley); многие мысли применимы также к Python. Несмотря на многочисленные синтаксические различия, на семантическом уровне Python и Ruby ближе друг к другу, чем к Java или C++.

В презентации «In Design Patterns in Dynamic Languages» (*http://norvig.com/design-patterns/*) (слайды) Петер Норвиг показывает, как с помощью полноправных функций (и других динамических средств) сделать некоторые классические паттерны более простыми или вообще ненужными.

За одно только введение к оригинальной книге Гамма и др. «Паттерны проектирования» можно было бы заплатить цену всей книги – это не просто каталог 23 паттернов, от очень важных до редко встречающихся. Введение – первоисточник часто цитируемых принципов проектирования: «Ставьте во главу угла интерфейс, а не реализацию» и «Предпочитайте композицию, а не наследование классов».

Применять паттерны к проектированию впервые предложил архитектор Кристофер Александер с соавторами в книге «A Pattern Language» (Oxford University Press). Идея Александера заключалась в том, чтобы создать стандартный словарь, который даст возможность разным коллективам обмениваться типичными решениями при проектировании зданий. М. Дж. Доминус создал завораживающий набор слайдов «'Design Patterns' Aren't», и Postscript-текст к ним, в котором доказывает, что оригинальное видение паттернов Александером более значительно, больше соответствует человеческой природе и применимо также к программной инженерии.

Поговорим

В языке Python есть полноправные функции и полноправные типы – средства, которые, согласно Норвигу, могут оказать влияние на 10 из 23 паттернов (слайд 10 из презентации «Design Patterns in Dynamic Languages» по адресу *http://norvig.com/design-patterns/*). В главе 9 мы видели, что в Python есть также обобщенные функции (раздел «Одиночная диспетчеризация и обобщенные функции»), похожие на мультиметоды из языка CLOS, которые в книге Гамма и др. названы более простым способом реализации классического паттерна Посетитель. Со своей стороны Норвиг утверждает, что мультиметоды упрощают паттерн Построитель (слайд 10). Адаптация паттернов к языку программирования – не точная наука.

На учебных курсах в разных уголках мира паттерны проектирования часто преподают на примерах из Java. Я не раз слышал от студентов, будто их заверяли в том, что оригинальные паттерны полезны в любом языке. Как оказалось, 23 «классических» паттерна из книги Гамма и др. прекрасно ложатся на «классический» Java, хотя первоначально излагались в основном в контексте C++ (в книге очень немного примеров для Smalltalk). Но это не значит, что каждый паттерн одинаково хорошо применим в любом языке. Авторы в самом начале

книги явно говорят, что «некоторые из наших паттернов напрямую поддерживаются менее распространенными объектно-ориентированными языками» (также еще раз прочитайте эпиграф к этой главе).

Библиография на тему паттернов проектирования в Python крайне скромна по сравнению с Java, C++ или Ruby. В разделе «Дополнительная литература» я упомянул книгу Gennadiy Zlobin «Learning Python Design Patterns», опубликованную только в ноябре 2013 года. А вот книга Russ Olsen «Design Patterns in Ruby» вышла еще в 2007 году и насчитывает 384 страницы – на 284 больше, чем работа Злобина.

Но будем надеяться, что с ростом популярности Python в академических кругах о паттернах проектирования в этом языке станут писать больше. Кроме того, в Java 8 появились ссылки на методы и анонимные функции – средства, которых ждали очень давно, – и есть надежда, что это стимулирует поиск новых подходов к паттернам в Java. В общем, надо признать, что языки развиваются, а вместе с ними и наши представления о том, как применять классические паттерны проектирования.

Зов (*call*) предков

Когда мы все вместе вносили в книгу последние исправления, технический рецензент Леонардо Рохаэль сделал удивительное открытие.

Если у функций есть метод `__call__`, а методы сами являются вызываемыми объектами, то, возможно, у метода `__call__` тоже есть метод `__call__`?

Не знаю, будет ли его открытие полезным, но факт остается фактом:

```
>>> def turtle():
...     return 'eggs'
...
>>> turtle()
'eggs'
>>> turtle.__call__()
'eggs'
>>> turtle.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__.__call__.__call__()
'eggs'
>>>
turtle.__call__.__call__.__call__.__call__.__call__.__call__()
'eggs'
>>>
turtle.__call__.__call__.__call__.__call__.__call__.__call__.__call__()
'eggs'
```

Черепахи – и нет им конца! (*https://en.wikipedia.org/wiki/Turtles_all_the_way_down*.)

Часть III

Классы и протоколы

Глава 11

Объект в духе Python

> Чтобы библиотека или каркас были питоническими, нужно сделать так, чтобы программист на Python мог максимально легко и естественно понять, как решать задачу.
>
> – Мартин Фаассен, автор каркасов на Python и JavsScript[1]

Благодаря модели данных в Python пользовательские типы могут вести себя так же естественно, как встроенные. И это достигается безо всякого наследования, в духе *утиной типизации*: достаточно просто реализовать методы, необходимые для того, чтобы объект вел себя ожидаемым образом.

В предыдущих главах мы рассказали о структуре и поведении многих встроенных объектов. А теперь займемся созданием собственных классов, которые ведут себя как настоящие объекты в Python. Вам не обязательно, да и не следует, реализовывать в своих прикладных классах столько специальных методов, сколько в примерах из этой главы. Но если вы пишете библиотеку или каркас, то программисты, которые будут пользоваться вашими классами, ожидают от них такого же поведения, как от классов, предоставляемых самими Python. Отвечать таким ожиданиям и значит соблюдать «дух Python».

Эта глава начинается с места, где закончилась глава 1, – мы покажем, как реализовать несколько специальных методов, которые обычно встречаются в объектах Python разных типов.

В этой главе мы узнаем, как:

- поддержать встроенные функции, которые порождают альтернативные представления объекта (`repr()`, `bytes()` и другие);
- реализовать альтернативный конструктор в виде метода класса;
- расширить мини-язык, используемый во встроенной функции `format()` и в методе `str.format()`;
- предоставить доступ к атрибутам только для чтения;
- сделать объект хешируемым, чтобы он мог быть элементом множества и ключом словаря;
- сэкономить память за счет использования `__slots__`.

Все это мы сделаем по мере разработки простого типа двумерного евклидова вектора `Vector2d`. Этот код ляжет в основу класса *N*-мерного вектора в главе 12.

По ходу дела мы дважды прервемся, чтобы обсудить два концептуально важных вопроса:

---

[1] Из статьи в блоге Фаассена «What is Pythonic?» (*https://blog.startifact.com/posts/older/what-is-pythonic.html*).

- как и когда использовать декораторы `@classmethod` и `@staticmethod`;
- закрытые и защищенные атрибуты в Python: использование, соглашения и ограничения.

Что нового в этой главе

Я заменил эпиграф и добавил несколько слов во второй абзац, посвященный вопросу о том, что такое «дух Python». В первом издании этот вопрос обсуждался в самом конце главы.

Раздел «Форматирование при выводе» дополнен материалом об f-строках, появившихся в версии Python 3.6. Это небольшое изменение, поскольку f-строки поддерживают тот же самый мини-язык форматирования, что встроенная функция `format()` и метод `str.format()`, поэтому все ранее написанные методы `__format__` будут работать и с f-строками.

Больше в главе почти ничего не изменилось – специальные методы в основном те же самые, что в Python 3.0, а основные идеи заложены еще в Python 2.2.

Начнем с методов представления объекта.

Представления объекта

В любом объектно-ориентированном языке есть по меньшей мере один стандартный способ получить строковое представление произвольного объекта. В Python таких способов два:

`repr()`

: Вернуть строку, представляющую объект в виде, удобном для разработчика. Это то, что мы видим, когда объект отображается на консоли Python или в отладчике.

`str()`

: Вернуть строку, представляющую объект в виде, удобном для пользователя. Это то, что печатает функция `print()`.

Как мы знаем из главы 1, для поддержки функций `repr()` и `str()` необходимо реализовать специальные методы `__repr__` и `__str__`.

Существует еще два специальных метода для поддержки альтернативных представлений объектов: `__bytes__` и `__format__`. Метод `__bytes__` аналогичен `__str__`: он вызывается функцией `bytes()`, чтобы получить представление объекта в виде последовательности байтов. А метод `__format__` вызывается f-строками, встроенной функцией `format()` и методом `str.format()` для получения строкового представления объектов с помощью специальных форматных кодов. В следующем разделе мы рассмотрим метод `__bytes__`, а вслед за ним метод `__format__`.

Если раньше вы программировали на Python 2, то имейте в виду, что в Python 3 методы `__repr__`, `__str__` и `__format__` всегда должны возвращать Unicode-строки (типа `str`). И лишь метод `__bytes__` должен возвращать последовательность байтов (типа `bytes`).

И снова класс вектора

Для демонстрации различных методов, генерирующих представления объектов, мы воспользуемся классом `Vector2d`, аналогичным рассмотренному в главе 1. В этом и следующих разделах мы будем постепенно наращивать его функциональность. В примере 11.1 показано базовое поведение, ожидаемое от объекта `Vector2d`.

Пример 11.1. У экземпляров `Vector2d` есть несколько представлений

```
>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y) ❶
3.0 4.0
>>> x, y = v1 ❷
>>> x, y
(3.0, 4.0) ❸
>>> v1
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1)) ❹
>>> v1 == v1_clone ❺
True
>>> print(v1) ❻
(3.0, 4.0)
>>> octets = bytes(v1) ❼
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1) ❽
5.0
>>> bool(v1), bool(Vector2d(0, 0)) ❾
(True,False)
```

❶ К компонентам `Vector2d` можно обращаться напрямую, как к атрибутам (методов чтения нет).
❷ Объект `Vector2d` можно распаковать в кортеж переменных.
❸ `repr` для объекта `Vector2d` имитирует исходный код конструирования экземпляра.
❹ Использование `eval` показывает, что результат `repr` для `Vector2d` – точное представление вызова конструктора[1].
❺ `Vector2d` поддерживает сравнение с помощью `==`; это полезно для тестирования.
❻ `print` вызывает функцию `str`, которая для `Vector2d` порождает упорядоченную пару.
❼ `bytes` пользуется методом `__bytes__` для получения двоичного представления.
❽ `abs` вызывает метод `__abs__`, чтобы вернуть модуль вектора.
❾ `bool` пользуется методом `__bool__`, чтобы вернуть `False` для объекта `Vector2d` нулевой длины, и `True` в противном случае.

Реализация класса `Vector2d` из примера 11.1 находится в файле *vector2d_v0.py* (пример 11.2). Код основан на примере 1.2, только методы для операций `+` и `*`

[1] Я использовал для клонирования объект `eval`, просто чтобы проиллюстрировать поведение `repr`; на практике клонировать объект проще и безопаснее с помощью функции `copy.copy`.

будут реализованы в главе 16. Мы также добавим метод для оператора `==`, поскольку он полезен для тестирования. В данный момент в `Vector2d` имеется несколько специальных методов для поддержки операций, которые питонист ожидает от хорошо спроектированного объекта.

Пример 11.2. vector2d_v0.py: пока что реализованы только специальные методы

```
from array import array
import math

class Vector2d:
    typecode = 'd' ❶

    def __init__(self, x, y):
        self.x = float(x) ❷
        self.y = float(y)

    def __iter__(self):
        return (i for i in (self.x, self.y)) ❸

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self) ❹

    def __str__(self):
        return str(tuple(self)) ❺

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) + ❻
                bytes(array(self.typecode, self))) ❼

    def __eq__(self, other):
        return tuple(self) == tuple(other) ❽

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

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

❶ `typecode` – это атрибут класса, которым мы воспользуемся, когда будем преобразовывать экземпляры `Vector2d` в последовательности байтов и наоборот.

❷ Преобразование `x` и `y` в тип `float` в методе `__init__` позволяет на ранней стадии обнаруживать ошибки, это полезно в случае, когда конструктор `Vector2d` вызывается с неподходящими аргументами.

❸ Наличие метода `__iter__` делает `Vector2d` итерируемым; именно благодаря ему работает распаковка (например, `x, y = my_vector`). Мы реализуем его просто с помощью генераторного выражения, которое отдает компоненты поочередно[1].

[1] Эту строку можно было бы записать и в виде `yield self.x; yield.self.y`. У меня еще найдется что сказать по поводу специального метода `__iter__`, генераторных выражений и ключевого слова `yield` в главе 17.

❹ Метод `__repr__` строит строку, интерполируя компоненты с помощью синтаксиса `{!r}` для получения их представления, возвращаемого функцией `repr`; поскольку `Vector2d` – итерируемый объект, `*self` поставляет компоненты `x` и `y` функции `format`.
❺ Из итерируемого объекта `Vector2d` легко построить кортеж для отображения в виде упорядоченной пары.
❻ Для генерации объекта типа `bytes` мы преобразуем `typecode` в `bytes` и конкатенируем ...
❼ ... с объектом `bytes`, полученным преобразованием массива, который построен путем обхода экземпляра.
❽ Для быстрого сравнения всех компонентов мы строим кортежи из операндов. Это работает, когда операнды являются экземплярами класса `Vector2d`, но не без проблем. См. предупреждение ниже.
❾ Модулем вектора называется длина гипотенузы прямоугольного треугольника с катетами `x` и `y`.
❿ Метод `__bool__` вызывает `abs(self)` для вычисления модуля, а затем преобразует полученное значение в тип `bool`, так что 0.0 преобразуется в `False`, а любое число, отличное от нуля, – в `True`.

Метод `__eq__` в примере 11.2 работает для операндов типа `Vector2d`, но возвращает `True` и в случае, когда экземпляр `Vector2d` сравнивается с другими итерируемыми объектами, содержащими точно такие же числовые значения (например, `Vector(3, 4) == [3, 4]`). Считать ли это ошибкой, зависит от точки зрения. Мы отложим дальнейшее обсуждение этого вопроса до главы 16, где рассматривается перегрузка операторов.

У нас имеется довольно полный набор базовых методов, но одного, очевидно, не хватает: восстановления объекта `Vector2d` из двоичного представления, порожденного функцией `bytes()`.

Альтернативный конструктор

Поскольку мы можем экспортировать `Vector2d` в виде последовательности байтов, хотелось бы иметь метод, который производит обратную операцию – конструирование `Vector2d` из двоичной последовательности. Заглянув в стандартную библиотеку в поисках источника вдохновения, мы обнаружим, что в классе `array.array` есть метод класса `.frombytes`, который нас вполне устраивает – мы видели его применение в разделе «Массивы» главы 2. Позаимствуем как имя, так и функциональность при написании метода класса `Vector2d` в файле *vector2d_v1.py* (пример 11.3).

Пример 11.3. Часть файла vector2d_v1.py: здесь показан только метод класса `frombytes`, добавленный в определение `Vector2d` из файла vector2d_v0.py (пример 11.2)

```
    @classmethod ❶
    def frombytes(cls, octets): ❷
        typecode = chr(octets[0]) ❸
        memv = memoryview(octets[1:]).cast(typecode) ❹
        return cls(*memv) ❺
```

❶ Метод класса снабжен декоратором `classmethod`.
❸ Аргумент `self` отсутствует; вместо него в аргументе `cls` передается сам класс.
❸ Прочитать `typecode` из первого байта.
❹ Создать объект `memoryview` из двоичной последовательности октетов и привести его к типу `typecode`[1].
❺ Распаковывать `memoryview`, образовавшийся в результате приведения типа, и получить пару аргументов, необходимых конструктору.

Поскольку мы только что воспользовались декоратором `classmethod`, весьма специфичным для Python, будет уместно сказать о нем несколько слов.

Декораторы classmethod и staticmethod

Декоратор `classmethod` не упоминается в пособии по Python, равно как и декоратор `staticmethod`. Те, кто изучал объектно-ориентированное программирование на примере Java, наверное, недоумевают, зачем в Python два декоратора, а не какой-нибудь один из них.

Начнем с `classmethod`. Его использование показано в примере 11.3: определить метод на уровне класса, а не отдельного экземпляра. Декоратор `classmethod` изменяет способ вызова метода таким образом, что в качестве первого аргумента передается сам класс, а не экземпляр. Типичное применение – альтернативные конструкторы, подобные `frombytes` из примера 11.3. Обратите внимание, как в последней строке метод `frombytes` использует аргумент `cls`, вызывая его для создания нового экземпляра: `cls(*memv)`.

Напротив, декоратор `staticmethod` изменяет метод так, что он не получает в первом аргументе ничего специального. По существу, статический метод – это просто обычная функция, определенная в теле класса, а не на уровне модуля. В примере 11.4 сравнивается работа `classmethod` и `staticmethod`.

Пример 11.4. Сравнение декораторов `classmethod` и `staticmethod`

```
>>> class Demo:
...     @classmethod
...     def klassmeth(*args): ❶
...         return args
...     @staticmethod
...     def statmeth(*args):
...         return args ❷
...
>>> Demo.klassmeth() ❸
(<class '__main__.Demo'>,)
>>> Demo.klassmeth('spam')
(<class '__main__.Demo'>, 'spam')
>>> Demo.statmeth() ❹
()
>>> Demo.statmeth('spam')
('spam',)
```

❶ `klassmeth` просто возвращает все позиционные аргументы.
❷ `statmeth` делает то же самое.

[1] Краткое введение в `memoryview`, где, в частности, описывается метод `.cast`, см. в разделе «Представления памяти» главы 2.

❸ Вне зависимости от способа вызова `Demo.klassmeth` получает класс `Demo` в качестве первого аргумента.
❹ `Demo.statmeth` ведет себя как обычная функция.

Декоратор `classmethod`, очевидно, полезен, но мне очень редко встречались хорошие примеры употребления `staticmethod`. Быть может, функция тесно связана с классом, хотя и не залезает в его «потроха», так что лучше разместить ее код поблизости. Но даже если так, размещение функции сразу до или после класса в том же модуле – это достаточно близко для любых практических целей[1].

Узнав, для чего применяется декоратор `classmethod` (и почему `staticmethod` не очень полезен), вернемся к вопросу о представлении объекта и посмотрим, как поддерживается форматирование вывода.

Форматирование при выводе

Встроенная функция `format()` и метод `str.format()` делегируют форматирование конкретному типу, вызывая его метод `.__format__(format_spec)`. Аргумент `format_spec` – это спецификатор формата, который либо:

- является вторым аргументом при вызове `format(my_obj, format_spec)`, либо
- равен тому, что находится после двоеточия в поле подстановки, обозначаемом скобками `{}` внутри f-строки или `fmt` при вызове `str.format()`.

Например:

```
>>> brl = 1 / 4.82 # курс бразильского реала к доллару США
>>> brl
0.20746887966804978
>>> format(brl, '0.4f') ❶
'0.2075'
>>> '1 BRL = {rate:0.2f} USD'.format(rate=brl) ❷
'1 BRL = 0.21 USD'
>>> f'1 USD = {1 / brl:0.2f} BRL' ❸
'1 USD = 4.82 BRL'
```

❶ Спецификатор формата `'0.4f'`.
❷ Спецификатор формата `'0.2f'`. Подстрока `'rate'` в поле подстановки называется именем поля. Она не связана со спецификатором формата, а определяет, какой аргумент метода `.format()` попадает в это поле подстановки.
❸ Снова спецификатор `'0.2f'`. Выражение `1 / brl` не является его частью.

Код, помеченный вторым маркером, – демонстрация важного момента: в форматной строке, например, `'{0.mass:5.3e}'` мы видим две совершенно разные нотации. Часть `'0.mass'` слева от двоеточия – это имя поля подстановки `field_name`, а часть `'5.3e'` после двоеточия – спецификатор формата. Нотация,

[1] Леонардо Рохаэль, один из технических рецензентов книги, не согласен с моим скептическим отношением к декоратору `staticmethod` и рекомендует прочитать статью в «The Definitive Guide on How to Use Static, Class or Abstract Methods in Python» (*https://julien.danjou.info/guide-python-static-class-abstract-methods/*) в блоге Жюльена Данжу, где приводятся контраргументы. Статья Данжу очень интересна, рекомендую ее. Но ее оказалось недостаточно, чтобы я изменил свое мнение о `staticmethod`. Решать вам.

применяемая в спецификаторе формата, называется также *мини-языком спецификации формата* (*https://docs.python.org/3/library/string.html#formatspec*).

Если вы раньше не встречались с f-строками, `format()` и `str.format()`, то хочу сказать, что мой опыт преподавания показывает, что лучше сначала изучить функцию `format()`, в которой используется только мини-язык спецификации формата. Освоив его, прочитайте разделы документации «Литералы форматной строки» (*https://docs.python.org/3/reference/lexical_analysis.html#f-strings*) и «Синтаксис форматной строки» (*https://docs.python.org/3/library/string.html#format-string-syntax*), чтобы разобраться с нотацией поля подстановки `{:}`, используемой в f-строках и методе `str.format()` (включая флаги преобразования `!s`, `!r` и `!a`). F-строки не отменяют метод `str.format()`: как правило, f-строки решают задачу, но иногда лучше задать форматную строку где-то в другом месте, а не там, где она используется для вывода.

Для нескольких встроенных типов в мини-языке спецификации формата предусмотрены специальные коды представления. Например, для типа `int` поддерживаются (среди прочих) коды `b` и `x`, обозначающие соответственно основание 2 и 16, а для типа `float` – код `f` для вывода значения с фиксированной точкой и `%` для вывода в виде процента:

```
>>> format(42, 'b')
'101010'
>>> format(2 / 3, '.1%')
'66.7%'
```

Мини-язык спецификации формата расширяемый, потому что каждый класс может интерпретировать аргумент `format_spec`, как ему вздумается. Например, классы из модуля `datetime` пользуются одними и теми же форматными кодами в функции `strftime()` и в своих методах `__format__`. Вот несколько примеров применения встроенной функции `format()` и метода `str.format()`:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> format(now, '%H:%M:%S')
'18:49:05'
>>> "It's now {:%I:%M %p}".format(now)
"It's now 06:49 PM"
```

Если в классе не реализован метод `__format__`, то используется метод, унаследованный от `object`, который возвращает значение `str(my_object)`. Поскольку в классе `Vector2d` есть метод `__str__`, это работает следующим образом:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
```

Но если передать спецификатор формата, то `object.__format__` возбудит исключение `TypeError`:

```
>>> format(v1, '.3f')
Traceback (most recent call last):
  ...
TypeError: non-empty format string passed to object.__format__
```

Исправим это, реализовав собственный мини-язык форматирования. Для начала предположим, что спецификатор формата, заданный пользователем, служит для форматирования каждой компоненты `float` вектора. Вот какой результат мы хотим получить:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

В примере 11.5 реализован метод `__format__`, дающий именно такой результат.

Пример 11.5. Метод `Vector2d.format`, попытка № 1

```
# в классе Vector2d

def __format__(self, fmt_spec=''):
    components = (format(c, fmt_spec) for c in self) ❶
    return '({}, {})'.format(*components) ❷
```

❶ Использовать встроенную функцию `format`, чтобы применить `fmt_spec` к каждой компоненте вектора и построить итерируемый объект, порождающий отформатированные строки.
❷ Подставить отформатированные строки в шаблон `'(x, y)'`.

Теперь добавим в наш мини-язык специальный форматный код: если спецификатор формата заканчивается буквой `'p'`, то будем отображать вектор в полярных координатах: `<r, θ>`, где `r` – модуль, а `θ` – угол в радианах. Остаток спецификатора формата (все, что предшествует `'p'`) используется, как и раньше.

При выборе буквы для специального форматного кода я стремился избегать совпадения с кодами для других типов. В мини-языке спецификации формата (*https://docs.python.org/3/library/string.html#formatspec*) для целых чисел используются коды `'bcdoxXn'`, для чисел с плавающей точкой – `'eEfFgGn%'`, а для строк – `'s'`. Поэтому для полярных координат я взял код `'p'`. Поскольку каждый класс интерпретирует коды независимо от остальных, использование одной и той же буквы в разных классах не является ошибкой, но может вызвать недоумение у пользователей.

Для вычисления полярных координат у нас уже есть метод `__abs__`, возвращающий модуль, а для получения угла напишем простой метод `angle`, в котором используется функция `math.atan2()`. Вот его код:

```
# в классе Vector2d
def angle(self):
    return math.atan2(self.y, self.x)
```

Теперь мы можем обобщить метод `__format__` для вывода представления в полярных координатах.

Пример 11.6. Метод `Vector2d.format`, попытка № 2 – теперь и в полярных координатах

```
    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('p'): ❶
            fmt_spec = fmt_spec[:-1] ❷
            coords = (abs(self), self.angle()) ❸
            outer_fmt = '<{}, {}>' ❹
        else:
            coords = self ❺
            outer_fmt = '({}, {})' ❻
        components = (format(c, fmt_spec) for c in coords) ❼
        return outer_fmt.format(*components) ❽
```

❶ Формат заканчивается буквой `'p'`: полярные координаты.
❷ Удалить суффикс `'p'` из `fmt_spec`.
❸ Построить кортеж полярных координат: `(magnitude, angle)`.
❹ Сконфигурировать внешний формат, используя угловые скобки.
❺ Иначе использовать компоненты `x`, `y` вектора `self` для представления в прямоугольных координатах.
❻ Сконфигурировать внешний формат, используя круглые скобки.
❼ Породить итерируемый объект, компонентами которого являются отформатированные строки.
❽ Подставить строки во внешний формат.

Ниже показаны результаты, полученные с помощью кода из примера 11.6.

```
>>> format(Vector2d(1, 1), 'p')
'<1.4142135623730951, 0.7853981633974483>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

Как видно из этого раздела, совсем несложно расширить мини-язык спецификации формата для поддержки пользовательских типов.

Теперь перейдем к вопросу, не относящемуся к видимому представлению: мы сделаем класс `Vector2d` хешируемым, чтобы можно было создавать множества векторов и использовать векторы в качестве ключей словаря.

Хешируемый класс Vector2d

До сих пор экземпляры класса `Vector2d` не были хешируемыми, поэтому мы не могли поместить их в множество:

```
>>> hash(v1)
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'Vector2d'
>>> set([v1])
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'Vector2d'
```

Чтобы класс `Vector2d` был хешируемым, мы должны реализовать метод `__hash__` (необходим еще метод `__eq__`, но он у нас уже есть). Нужно также, чтобы векто-

ры были неизменяемыми, как было сказано во врезке «Что значит “хешируемый”?» в главе 3.

Пока ничто не мешает любому пользователю написать `v1.x = 7`, т. к. нигде в коде не говорится, что изменение `Vector2d` запрещено. Вот какое поведение мы хотим получить:

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 7
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
```

Мы добьемся этого, сделав компоненты `x` и `y` свойствами, доступными только для чтения.

Пример 11.7. vector2d_v3.py: показаны только изменения, необходимые, чтобы сделать класс `Vector2d` неизменяемым, полный листинг см. в примере 11.11

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x) ❶
        self.__y = float(y)

    @property ❷
    def x(self): ❸
        return self.__x ❹

    @property ❺
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y)) ❻

    # остальные методы: такие же, как в предыдущем варианте Vector2d
```

❶ Использовать ровно два начальных подчерка (и нуль или один конечный), чтобы сделать атрибут закрытым[1].
❷ Декоратор `@property` помечает метод чтения свойства.
❸ Метод чтения назван так же, как соответствующее открытое свойство: `x`.
❹ Просто вернуть `self.__x`.
❺ Повторяем то же самое для свойства `y`.
❻ Все методы, которые просто читают компоненты `x` и `y`, не изменяются, только теперь `self.x` и `self.y` означает чтение открытых свойств, а не закрытых атрибутов. Поэтому оставшаяся часть класса не показана.

[1] Плюсы и минусы закрытых атрибутов – тема раздела «Закрытые и защищенные атрибуты в Python» ниже в этой главе.

`Vector.x` и `Vector.y` – примеры свойств, доступных только для чтения. Свойства, доступные для чтения и записи, рассматриваются в главе 22, где мы детально изучим декоратор `@property`.

Теперь, когда векторы стали неизменяемыми, мы можем реализовать метод `__hash__`. Он должен возвращать `int` и в идеале учитывать хеши объектов-атрибутов, которые используются также в методе `__eq__`, потому что у равных объектов хеши также должны быть одинаковы. В документации по специальному методу `__hash__` (*https://docs.python.org/3/reference/datamodel.html*) рекомендуется вычислять хеш кортежа покомпонентно, так мы и поступим (см. пример 11.8).

Пример 11.8. vector2d_v3.py: реализация хеширования

```
    # в классе Vector2d:
    def __hash__(self):
        return hash((self.x, self.y))
```

После добавления метода `__hash__` мы получили хешируемые векторы:

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)
(1079245023883434373, 1994163070182233067)
>>> {v1, v2}
{Vector2d(3.1, 4.2), Vector2d(3.0, 4.0)}
```

Строго говоря, для создания хешируемого типа необязательно вводить свойства или как-то иначе защищать атрибуты экземпляра от изменения. Требуется только корректно реализовать методы `__hash__` и `__eq__`. Но значение хешируемого объекта никогда не должно изменяться, так что представился отличный повод поговорить о свойствах, доступных только для чтения.

Если вы собираетесь создать тип с разумным скалярным числовым значением, то имеет смысл реализовать также методы `__int__` и `__float__`, которые вызываются из конструкторов `int()` и `float()`, используемых в некоторых контекстах для приведения типов. Существует также метод `__complex__`, поддерживающий встроенный конструктор `complex()`. Быть может, в классе `Vector2d` и стоило бы реализовать метод `__complex__`, но это я оставляю вам в качестве упражнения.

Поддержка позиционного сопоставления с образцом

До сих пор экземпляры `Vector2d` были совместимы с именованными классами-образцами, которые рассматривались в разделе «Именованные классы-образцы» главы 5.

В примере 11.9 все эти именованные образцы работают в полном соответствии с ожиданиями.

Пример 11.9. Именованные образцы для субъектов типа `Vector2d` – требуется версия Python 3.10

```
def keyword_pattern_demo(v: Vector2d) -> None:
    match v:
        case Vector2d(x=0, y=0):
            print(f'{v!r} is null')
        case Vector2d(x=0):
            print(f'{v!r} is vertical')
        case Vector2d(y=0):
            print(f'{v!r} is horizontal')
        case Vector2d(x=x, y=y) if x==y:
            print(f'{v!r} is diagonal')
        case _:
            print(f'{v!r} is awesome')
```

Однако при попытке использовать позиционный образец:

```
case Vector2d(_, 0):
    print(f'{v!r} is horizontal')
```

мы получаем:

```
TypeError: Vector2d() accepts 0 positional sub-patterns (1 given)
```

Чтобы заставить `Vector2d` работать с позиционными образцами, необходимо добавить атрибут класса `__match_args__`, перечислив в нем атрибуты экземпляра в том порядке, в каком они будут использоваться при позиционном сравнении с образцом:

```
class Vector2d:
    __match_args__ = ('x', 'y')

    # и т. д.
```

Теперь мы можем сэкономить несколько нажатий клавиш при написании образцов, которые сопоставляются с субъектами `Vector2d`, как показано в примере 11.10.

Пример 11.10. Позиционные образцы для субъектов типа `Vector2d` – требуется версия Python 3.10

```
def positional_pattern_demo(v: Vector2d) -> None:
    match v:
        case Vector2d(0, 0):
            print(f'{v!r} is null')
        case Vector2d(0):
            print(f'{v!r} is vertical')
        case Vector2d(_, 0):
            print(f'{v!r} is horizontal')
        case Vector2d(x, y) if x==y:
            print(f'{v!r} is diagonal')
        case _:
            print(f'{v!r} is awesome')
```

Атрибут класса `__match_args__` не обязан включать все открытые атрибуты экземпляра. В частности, если у метода `__init__` имеются обязательные и факультативные аргументы, которые присваиваются атрибутам экземпляра, то, воз-

можно, будет разумно перечислить в `__match_args__` обязательные аргументы, но опустить факультативные.

Теперь сделаем шаг назад и окинем взглядом все, что мы написали в классе `Vector2d`.

Полный код класса Vector2d, версия 3

По ходу работы над классом `Vector2d` мы показывали только фрагменты кода, а в примере 11.11 представлен полный код *vector2d_v3.py* со всеми тестами, которые я писал, пока разрабатывал его.

Пример 11.11. vector2d_v3.py: полный код

```
«»»
Класс двумерного вектора

    >>> v1 = Vector2d(3, 4)
    >>> print(v1.x, v1.y)
    3.0 4.0
    >>> x, y = v1
    >>> x, y
    (3.0, 4.0)
    >>> v1
    Vector2d(3.0, 4.0)
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0)
    >>> octets = bytes(v1)
    >>> octets

b'd\x00\x00\x00\x00\x00\x00\x08@\x00\x00\x00\x00\x00\x00\x
00\x10@'
    >>> abs(v1)
    5.0
    >>> bool(v1), bool(Vector2d(0, 0))
    (True, False)


Тест метода класса ``.frombytes()``:

    >>> v1_clone = Vector2d.frombytes(bytes(v1))
    >>> v1_clone
    Vector2d(3.0, 4.0)
    >>> v1 == v1_clone
    True


Тесты ``format()`` с декартовыми координатами:

    >>> format(v1)
    '(3.0, 4.0)'
    >>> format(v1, '.2f')
    '(3.00, 4.00)'
```

```
    >>> format(v1, '.3e')
    '(3.000e+00, 4.000e+00)'


Тесты метода ``angle``::

    >>> Vector2d(0, 0).angle()
    0.0
    >>> Vector2d(1, 0).angle()
    0.0
    >>> epsilon = 10**-8
    >>> abs(Vector2d(0, 1).angle() - math.pi/2) < epsilon
    True
    >>> abs(Vector2d(1, 1).angle() - math.pi/4) < epsilon
    True


Тесты ``format()`` с полярными координатами:

    >>> format(Vector2d(1, 1), 'p') # doctest:+ELLIPSIS
    '<1.414213..., 0.785398...>'
    >>> format(Vector2d(1, 1), '.3ep')
    '<1.414e+00, 7.854e-01>'
    >>> format(Vector2d(1, 1), '0.5fp')
    '<1.41421, 0.78540>'


Тесты свойств `x` и `y`, доступных только для чтения:

    >>> v1.x, v1.y
    (3.0, 4.0)
    >>> v1.x = 123
    Traceback (most recent call last):
      ...
    AttributeError: can't set attribute 'x'


Тесты хеширования:

    >>> v1 = Vector2d(3, 4)
    >>> v2 = Vector2d(3.1, 4.2)
    >>> len({v1, v2})
    2

"""

from array import array
import math

class Vector2d:
    __match_args__ = ('x', 'y')

    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)
```

```
    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(array(self.typecode, self)))

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __hash__(self):
        return hash((self.x, self.y))

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

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

    def angle(self):
        return math.atan2(self.y, self.x)

    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('p'):
            fmt_spec = fmt_spec[:-1]
            coords = (abs(self), self.angle())
            outer_fmt = '<{}, {}>'
        else:
            coords = self
            outer_fmt = '({}, {})'
        components = (format(c, fmt_spec) for c in coords)
        return outer_fmt.format(*components)

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(*memv)
```

Подведем итоги. В этом и предыдущем разделах мы видели некоторые специальные методы, которые должен иметь полноценный объект.

Реализовывать эти специальные методы следует, только если приложение в них нуждается. Пользователям наплевать, соответствует ваш объект «духу Python» или нет.

С другой стороны, если ваши классы являются частью библиотеки, предназначенной для других программистов, то вы не знаете, что они будут делать с вашими объектами, так что лучше, если они будут отвечать духу Python.

Представленный в примере 11.11 класс `Vector2d` – написанный в педагогических целях код, изобилующий специальными методами, относящимися к представлению объекта, а не образец для создания любого пользовательского класса.

В следующем разделе мы отвлечемся от класса `Vector2d` и обсудим дизайн и недостатки механизма закрытых атрибутов в Python – двойное подчеркивание в начале имени `self.__x`.

Закрытые и «защищенные» атрибуты в Python

В Python не существует способа создать закрытые переменные, как с помощью модификатора `private` в Java. Мы имеем лишь простой механизм, предотвращающий случайную модификацию «закрытого» атрибута в подклассе.

Рассмотрим такую ситуацию: кто-то написал класс `Dog`, где используется внутренний атрибут экземпляра `mood`, который автор не хотел раскрывать клиентам. Нам нужно написать подкласс `Dog` – `Beagle`. Если мы создадим свой атрибут экземпляра `mood`, не подозревая о конфликте имен, то затрем атрибут `mood`, используемый в методах, унаследованных от `Dog`. Отлаживать такую ошибку непросто.

Чтобы предотвратить это, мы можем назвать атрибут `__mood` (с двумя начальными подчерками и, возможно, одним – не более – конечным подчерком). Тогда Python сохранит имя в словаре экземпляра `__dict__`, добавив в начало один подчерк и имя класса, т. е. в классе `Dog` атрибут `__mood` будет называться `_Dog__mood`, а в классе `Beagle` – `_Beagle__mood`. Эта особенность языка имеет прелестное название – *декорирование имен* (name mangling).

В примере 11.12 показано, как это выглядит в классе `Vector2d` из примера 11.7.

Пример 11.12. Имена закрытых атрибутов «декорируются» добавлением префикса _ и имени класса

```
>>> v1 = Vector2d(3, 4)
>>> v1.__dict__
{'_Vector2d__y': 4.0, '_Vector2d__x': 3.0}
>>> v1._Vector2d__x
3.0
```

Декорирование имен служит скорее предохранительной мерой, а не средством обеспечения безопасности; идея в том, чтобы предотвратить случайный доступ, а не намеренное желание причинить зло (на рис. 11.1 изображено еще одно предохранительное устройство).

Любой программист, знающий, как устроено декорированное имя, может напрямую прочитать закрытый атрибут, как показано в последней строке при-

мера 11.12, – и это полезно для отладки и сериализации. Можно также присвоить значение закрытому компоненту `Vector2d`, просто написав `v1._Vector__x = 7`. Но если вы так сделаете в производственном коде, то жаловаться на то, что программа перестала работать, будет некому.

Декорирование имен нравится далеко не всем питонистам, как и неприглядные имена вида `self.__x`. Некоторые предпочитают вместо этого добавлять одиночный подчерк, чтобы «защитить» атрибуты соглашением (например, `self._x`). Критики автоматического декорирования имен с двумя начальными подчерками говорят, что проблему случайного затирания атрибутов следует решать с помощью соглашений об именовании. Ян Байкинг, автор pip, virtualenv и других проектов, пишет:

> Ни в коем случае не используйте два подчерка в начале. Это приватно до безобразия. Если конфликт имен представляет проблему, применяйте явное декорирование (например, _MyThing_blahblah). По сути дела, это то же самое, что два подчерка, только делает прозрачным то, что два подчерка скрывают[1].

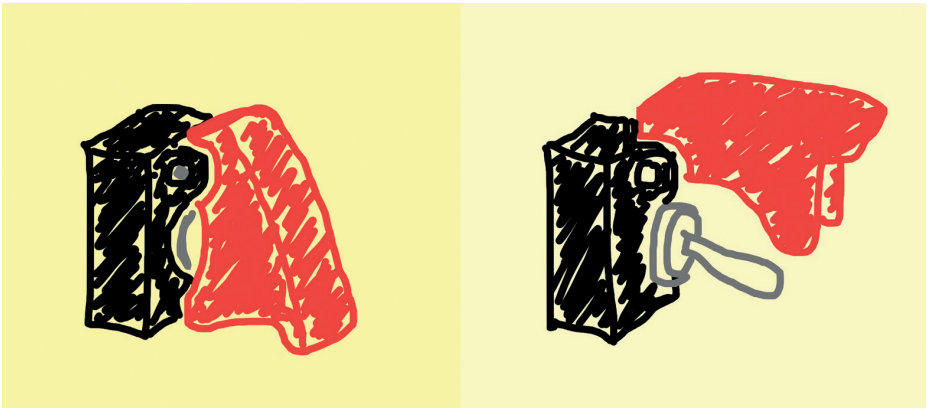


Рис. 11.1. Крышка рубильника – это предохранительное устройство, не гарантирующее безопасность, она предотвращает случайное, а не злонамеренное включение

Одиночный начальный подчерк в именах атрибутов не означает ничего особенного для интерпретатора Python, но в среде программистов, пишущих на этом языке, бытует соглашение о том, что не надо обращаться к таким атрибутам извне самого класса[2]. Нетрудно уважать право на приватность объекта, который помечает свои атрибуты одиночным знаком _, равно как и соблюдать соглашение о том, что переменные с именами, состоящими только из заглавных букв (`ALL_CAPS`), должны считаться константами.

---

[1] Из «Руководства по стилю программирования в Paste» (*http://pythonpaste.org/StyleGuide.html*).

[2] В модулях одиночный подчерк в начале имени верхнего уровня имеет специальный смысл: если написать `from mymod import *`, то имена с префиксом _ не будут импортироваться из `mymod`. Но ничто не мешает явно написать `from mymod import _privatefunc`. Это объясняется в «Учебном пособии по Python» в разделе 6.1 «Еще о модулях» (*https://docs.python.org/3/tutorial/modules.html#more-on-modules*).

Атрибуты с одиночным подчерком в начале в некоторых уголках документации Python называются «защищенными»[1]. Практика «защиты» атрибутов соглашением вида `self._x` широко распространена, но название «защищенный» не столь употребительно. Некоторые даже называют такие атрибуты «закрытыми».

Подведем итог: компоненты класса `Vector2d` «закрыты», а экземпляры `Vector2d` «неизменяемы». Устрашающие кавычки поставлены, потому что на самом деле не существует способа сделать их по-настоящему закрытыми и неизменяемыми[2].

Но вернемся к классу `Vector2d`. В последнем разделе мы рассмотрим специальный атрибут (не метод) `__slots__`, который влияет на внутреннее хранение данных в объекте, что может заметно сократить потребление памяти, но почти не сказывается на открытом интерфейсе класса.

Экономия памяти с помощью атрибута класса __slots__

По умолчанию Python хранит атрибуты экземпляра в словаре `__dict__`, принадлежащем самому экземпляру. В разделе «Практические последствия механизма работы dict» главы 3 мы видели, что со словарями сопряжены значительные накладные расходы – даже при всех упомянутых там оптимизациях. Но если определить атрибут класса `__slots__`, в котором хранится последовательность имен атрибутов, то Python будет использовать альтернативную модель хранения для атрибутов экземпляра: атрибуты с именами, представленными в `__slots__`, хранятся в скрытом массиве ссылок, потребляющем намного меньше памяти, чем `dict`. Посмотрим, как это работает на простых примерах.

Пример 11.13. Класс `Pixel` использует `__slots__`

```
>>> class Pixel:
...     __slots__ = ('x', 'y') ❶
...
>>> p = Pixel() ❷
>>> p.__dict__ ❸
Traceback (most recent call last):
  ...
AttributeError: 'Pixel' object has no attribute '__dict__'
>>> p.x = 10 ❹
>>> p.y = 20 ❺
>>> p.color = 'red'
Traceback (most recent call last):
  ...
AttributeError: 'Pixel' object has no attribute 'color'
```

❶ `__slots__` должен присутствовать в момент создания класса; добавление или изменение впоследствии не оказывает никакого эффекта. Имена атрибутов могут храниться в кортеже или в списке, но лично я предпочитаю кортеж, чтобы сразу было понятно, что изменять его нет никакого смысла.

---

[1] Один такой пример – документация по модулю gettext (*https://docs.python.org/3/library/gettext.html# gettext.NullTranslations*).

[2] Если такое состояние дел вас угнетает и вызывает желание сделать Python больше похожим в этом отношении на Java, не читайте мои высказывания по поводу относительности возможностей модификатора `private` в Java во врезке «Поговорим».

❷ Создать экземпляр `Pixel`, потому что мы хотим видеть, как `__slots__` влияет на экземпляры.
❸ Первый эффект: у экземпляров `Pixel` нет атрибута `__dict__`.
❸ Установить атрибуты `p.x` и `p.y` как обычно.
❹ Второй эффект: попытка установить атрибут, отсутствующий в `__slots__`, приводит к исключению `AttributeError`.

Пока все хорошо. А теперь давайте создадим подкласс `Pixel` в примере 11.14, чтобы увидеть сторону `__slots__`, противоречащую интуиции.

Пример 11.14. Класс `OpenPixel` является подклассом `Pixel`

```
>>> class OpenPixel(Pixel): ❶
...     pass
...
>>> op = OpenPixel()
>>> op.__dict__ ❷
{}
>>> op.x = 8 ❸
>>> op.__dict__ ❹
{}
>>> op.x ❺
8
>>> op.color = 'green' ❻
>>> op.__dict__ ❼
{'color': 'green'}
```

❶ `OpenPixel` не объявляет собственных атрибутов.
❷ Сюрприз: у экземпляров `OpenPixel` есть атрибут `__dict__`.
❸ Если установить атрибут `x` (присутствующий в атрибуте `__slots__` базового класса `Pixel`) ...
❹ ... то он не появляется в атрибуте `__dict__` экземпляра ...
❺ ... но сохраняется в скрытом массиве ссылок в этом экземпляре.
❻ Если установить атрибут, отсутствующий в `__slots__` ...
❼ ... то он сохраняется в атрибуте `__dict__` экземпляра.

Пример 11.14 показывает, что эффект `__slots__` лишь частично наследуется подклассом. Чтобы гарантировать отсутствие `__dict__` в подклассе, необходимо еще раз объявить в нем `__slots__`.

Если объявить `__slots__ = ()` (пустой кортеж), то у экземпляров подкласса не будет `__dict__` и они будут принимать только атрибуты, перечисленные в атрибуте `__slots__` базового класса.

Если вы хотите, чтобы в подклассе не было дополнительных атрибутов, то перечислите имена тех, что вас интересуют, в `__slots__`, как показано в примере 11.15.

Пример 11.15. `ColorPixel`, еще один подкласс `Pixel`

```
>>> class ColorPixel(Pixel):
...     __slots__ = ('color',) ❶
>>> cp = ColorPixel()
>>> cp.__dict__ ❷
Traceback (most recent call last):
...
```

```
AttributeError: 'ColorPixel' object has no attribute '__dict__'
>>> cp.x = 2
>>> cp.color = 'blue' ❸
>>> cp.flavor = 'banana'
Traceback (most recent call last):
  ...
AttributeError: 'ColorPixel' object has no attribute 'flavor'
```

❶ Атрибуты `__slots__` суперклассов добавляются в `__slots__` текущего класса. Не забывайте, что кортежи с одним элементом должны заканчиваться запятой.
❷ Экземпляры `ColorPixel` не имеют атрибута `__dict__`.
❸ Мы можем установить только атрибуты, объявленные в `__slots__` этого класса и его суперклассов, и больше никакие.

Можно и «память сэкономить, и косточкой не подавиться»: если добавить имя `'__dict__'` в список `__slots__`, то все атрибуты, перечисленные в `__slots__`, будут храниться в массиве ссылок, принадлежащем экземпляру, но при этом разрешено динамически создавать новые атрибуты, которые хранятся в словаре `__dict__`, как обычно. Это обязательно, если вы хотите пользоваться декоратором `@cached_property` (рассматривается в разделе «Шаг 5: кеширование свойств с помощью functools» главы 22).

Разумеется, помещение `'__dict__'` в атрибут `__slots__` может свести на нет все преимущества последнего, но это зависит от количества статических и динамических атрибутов и того, как они используются. Бездумная оптимизация еще хуже преждевременной: вы увеличиваете сложность, а взамен не получаете ничего.

Существует еще один специальный атрибут экземпляра, который имеет смысл сохранить: `__weakref__` необходим, чтобы объект поддерживал слабые ссылки (см. раздел «Слабые ссылки» главы 6). По умолчанию этот атрибут присутствует в экземплярах всех пользовательских классов. Однако если в классе определен атрибут `__slots__`, а вам нужно, чтобы его экземпляры могли быть объектами слабых ссылок, то `'__weakref__'` необходимо явно включить в список имен атрибутов в `__slots__`.

Теперь посмотрим, какое влияние оказывает атрибут `__slots__` на класс `Vector2d`.

Простое измерение экономии, достигаемой за счет __slot__

В примере 11.16 показана реализация `__slots__` в классе `Vector2d`.

Пример 11.16. vector2d_v3_slots.py: по сравнению с версией `Vector2d` добавился только атрибут `__slots__`

```
class Vector2d:
    __match_args__ = ('x', 'y') ❶
    __slots__ = ('__x', '__y') ❷

    typecode = 'd'
    # методы такие же, как в предыдущей версии
```

❶ В `__match_args__` перечислены открытые атрибуты для позиционного сопоставления с образцом.

❷ Напротив, в `__slots__` перечислены имена атрибутов экземпляра, которые в данном случае являются закрытыми.

Чтобы измерить экономию памяти, я написал скрипт *mem_test.py*. Он принимает имя модуля, содержащего вариант класса `Vector2d`, и с помощью спискового включения строит список с 10 000 000 экземпляров `Vector2d`. При первом прогоне (пример 11.17) я взял класс `vector2d_v3.Vector2d` (из примера 11.7), а при втором – версию класса с атрибутом `__slots__` из примера 11.16.

Пример 11.17. mem_test.py создает 10 миллионов экземпляров класса `Vector2d` из указанного при запуске модуля

```
$ time python3 mem_test.py vector2d_v3
Selected Vector2d type: vector2d_v3.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      6,983,680
  Final RAM usage:  1,666,535,424

real    0m11.990s
user    0m10.861s
sys     0m0.978s
$ time python3 mem_test.py vector2d_v3_slots
Selected Vector2d type: vector2d_v3_slots.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      6,995,968
  Final RAM usage:    577,839,104

real    0m8.381s
user    0m8.006s
sys     0m0.352s
```

Как видно из примера 11.17, потребление памяти составляет 1,55 ГиБ при использовании в каждом из 10 миллионов экземпляров `Vector2d` словаря `__dict__`, но снижается до 551 МиБ, если используется атрибут `__slots__`. К тому же версия с `__slots__` еще и быстрее. Скрипт *mem_test.py* просто загружает модуль, измеряет потребление памяти и красиво выводит результаты. Его код можно найти по адресу *https://github.com/fluentpython/example-code-2e/blob/master/11-pythonic-obj/mem_test.py*.

При работе с миллионами объектов, содержащих числовые данные, следует использовать массивы NumPy (см. раздел «NumPy» главы 2), которые не только эффективно расходуют память, но и располагают оптимизированными функциями, в том числе применяемыми к массиву в целом. Я проектировал класс `Vector2d` только для того, чтобы было на чем обсуждать специальные методы, т. к. стараюсь по возможности избегать бессмысленных примеров с `foo` и `bar`.

Проблемы при использовании __slots__

Атрибут `__slots__` при правильном использовании может дать значительную экономию памяти, но есть несколько подводных камней.

❍ Не забывайте заново объявлять `__slots__` в каждом подклассе, потому что унаследованный атрибут интерпретатор игнорирует.

- Экземпляры класса могут иметь только атрибуты, явно перечисленные в `__slots__`, если в него не включено также имя `'__dict__'` (однако при этом вся экономия памяти может быть сведена на нет).
- Для классов, где используется `__slots__`, нельзя употреблять декоратор `@cached_property`, если только в `__slots__` явно не включено имя `'__dict__'`.
- Экземпляры класса не могут быть объектами слабых ссылок, если не включить в `__slots__` имя `'__weakref__'`.

Последняя тема этой главы – переопределение атрибутов класса в экземплярах и подклассах.

Переопределение атрибутов класса

Отличительной особенностью Python является использование атрибутов класса в качестве значений по умолчанию для атрибутов экземпляра. В классе `Vector2d` имеется атрибут класса `typecode`. Он дважды используется в методе `__bytes__`, но там мы осознанно писали `self.typecode`. Поскольку экземпляры класса `Vector2d` создаются без собственного атрибута `typecode`, значение `self.typecode` по умолчанию берется из атрибута класса `Vector2d.typecode`.

Но если мы упоминаем в коде имя несуществующего атрибута, то создается новый атрибут экземпляра, например `typecode`, а одноименный атрибут класса остается без изменения. Однако начиная с этого момента всякий раз, как код, работающий с этим экземпляром, видит `self.typecode`, читается атрибут `typecode` экземпляра, т. е. атрибут класса с тем же именем маскируется. Это открывает возможность настроить отдельный экземпляр, изменив в нем `typecode`.

По умолчанию `Vector2d.typecode` равен `'d'`, т. е. при экспорте в тип `bytes` каждая компонента вектора представляется 8-байтовым числом с плавающей точкой двойной точности. Если же перед экспортом присвоить атрибуту `typecode` конкретного экземпляра `Vector2d` значение `'f'`, то каждая компонента будет экспортироваться в виде 4-байтового числа с плавающей точкой одинарной точности. См. пример 11.18.

Поскольку мы обсуждаем динамическое добавление атрибута, то в примере 11.18 используется реализация `Vector2d` без `__slots__`, показанная в примере 11.11.

Пример 11.18. Настройка экземпляра путем установки атрибута `typecode`, первоначально унаследованного от класса

```
>>> from vector2d_v3 import Vector2d
>>> v1 = Vector2d(1.1, 2.2)
>>> dumpd = bytes(v1)
>>> dumpd
b'd\x9a\x99\x99\x99\x99\xf1?\x9a\x99\x99\x99\x99\x01@'
>>> len(dumpd) ❶
17
>>> v1.typecode = 'f' ❷
>>> dumpf = bytes(v1)
>>> dumpf
```

```
b'f\xcd\xcc\x8c?\xcd\xcc\x0c@'
>>> len(dumpf) ❸
9
>>> Vector2d.typecode ❹
'd'
```

❶ Подразумеваемое по умолчанию представление `bytes` имеет длину 17 байт.
❷ Присвоить `typecode` значение `'f'` в экземпляре `v1`.
❸ Теперь длина представления в виде `bytes` составляет 9 байт.
❹ `Vector2d.typecode` не изменился; атрибут `typecode` равен `'f'` только в экземпляре `v1`.

Теперь должно быть понятно, почему при экспорте объекта `Vector2d` в формате `bytes` результирующее представление начинается с `typecode`: мы хотели поддержать различные форматы экспорта.

Если вы хотите изменить сам атрибут класса, то должны присвоить ему значение напрямую, а не через экземпляр. Чтобы изменить значение `typecode` по умолчанию, распространяющееся на все экземпляры (не имеющие собственного атрибута `typecode`), нужно написать:

```
>>> Vector2d.typecode = 'f'
```

Однако существует идиоматический способ добиться более постоянного эффекта и явно выразить смысл изменения. Поскольку атрибуты класса открыты и наследуются подклассами, то принято настраивать атрибут класса в подклассе. В основанных на классах представлениях Django эта техника применяется сплошь и рядом. Она демонстрируется в примере 11.19.

Пример 11.19. `ShortVector2d` – подкласс `Vector2d`, единственное отличие которого – переопределение атрибута `typecode` по умолчанию

```
>>> from vector2d_v3 import Vector2d
>>> class ShortVector2d(Vector2d): ❶
...     typecode = 'f'
...
>>> sv = ShortVector2d(1/11, 1/27) ❷
>>> sv
ShortVector2d(0.09090909090909091, 0.037037037037037035) ❸
>>> len(bytes(sv)) ❹
9
```

❶ Создать `ShortVector2d` как подкласс `Vector2d` только для того, чтобы переопределить атрибут класса `typecode`.
❷ Создать экземпляр `ShortVector2d` – объект `sv`.
❸ Инспектировать представление `sv`.
❹ Проверить, что экспортировано 9 байт, а не 17, как раньше.

Этот пример также объясняет, почему я не стал «зашивать» значение `class_name` в код `Vecto2d.__repr__`, а получаю его в виде `type(self).__name__`:

```
    # в классе Vector2d:

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)
```

Если бы я зашил `class_name`, то подклассы `Vector2d` и, в частности, `ShortVector2d` должны были бы переопределять метод `__repr__` только для того, чтобы изменить `class_name`. А, получая имя от функции `type`, примененной к экземпляру, я сделал `__repr__` безопасным относительно наследования.

На этом завершается рассмотрение реализации простого класса, который ведет себя, как положено в Python, пользуясь средствами, предоставляемыми моделью данных: предлагает различные представления объекта, реализует специализированный код форматирования, раскрывает атрибуты, доступные только для чтения, и поддерживает метод `hash()` для интеграции со множествами и отображениями.

Резюме

Целью этой главы была демонстрация специальных методов и соглашений в процессе разработки класса Python, который ведет себя ожидаемо.

Можно ли сказать, что реализация в файле *vector2d_v3.py* (пример 11.11) лучше соответствует духу Python, чем та, что находится в файле *vector2d_v0.py* (пример 11.2)? Конечно, в классе `Vector2d` из файла *vector2d_v3.py* задействовано больше механизмов Python. Но какую версию считать более идиоматичной, зависит от контекста использования. В «Дзен Python» Тима Питера сказано:

> Простое лучше, чем сложное.

Объект должен быть настолько простым, насколько возможно при соблюдении требований, – а не выставкой языковых средств. Если код является частью приложения, то следует поставить во главу угла то, что необходимо для поддержки конечных пользователей, – и не больше. Если код входит в библиотеку, предназначенную другим программистам, то будет разумно реализовать специальные методы, поддерживающие виды поведения, ожидаемые питонистами. Например, метод `__eq__`, возможно, не нужен с точки зрения бизнес-требований, но его наличие облегчает тестирование класса.

Развивая код `Vector2d`, я ставил себе целью предложить контекст для обсуждения специальных методов и соглашений о кодировании. В примерах из этой главы продемонстрировано несколько специальных методов, с которыми мы впервые встретились в табл. 1.1:

- методы строкового и байтового представления: `__repr__`, `__str__`, `__format__` и `__bytes__`;
- методы преобразования объекта в число: `__abs__`, `__bool__`, `__hash__`;
- оператор `__eq__` для тестирования преобразования в `bytes` и поддержки хеширования (наряду с методом `__hash__`).

Обеспечивая поддержку преобразования в `bytes`, мы заодно реализовали альтернативный конструктор `Vector2d.frombytes()` и попутно получили предлог для обсуждения декораторов `@classmethod` (очень полезного) и `@staticmethod` (не столь полезного, поскольку функции уровня модуля проще). Идея метода `frombytes` позаимствована у его тезки из класса `array.array`.

Мы видели, что мини-язык спецификации формата (*https://docs.python.org/3/library/string.html#formatspec*) можно расширить путем реализации метода `__format__`, который осуществляет несложный разбор строки `format_spec`, передавае-

мой встроенной функции `format(obj, format_spec)` или включенной в поле подстановки `'{:«format_spec»}'` в f-строках или строках, используемых в методе `str.format`.

Прежде чем сделать экземпляры класса `Vector2d` хешируемыми, мы постарались обеспечить их неизменяемость или, по крайней мере, предотвратить случайное изменение. Для этого мы сделали атрибуты `x` и `y` закрытыми и предоставили к ним доступ через свойства, доступные только для чтения. Затем реализовали метод `__hash__`, применяя рекомендуемую технику: объединить хеши атрибутов экземпляра с помощью оператора ИСКЛЮЧАЮЩЕЕ ИЛИ.

Далее мы обсудили экономию памяти, достигаемую с помощью атрибута `__slots__` в классе `Vector2d`, и опасности, подстерегающие на этом пути. Поскольку с использованием `__slots__` сопряжены некоторые сложности, делать это имеет смысл только при работе с очень большим количеством экземпляров – порядка миллионов, а не тысяч. В таких случаях часто лучше прибегнуть к библиотеке pandas (*https://pandas.pydata.org/*).

И напоследок мы обсудили вопрос о переопределении атрибута класса при доступе через экземпляры (например, `self.typecode`). Для этого мы сначала создали атрибут конкретного экземпляра, а затем породили подкласс и переопределили в нем атрибут на уровне класса.

В этой главе я не раз отмечал, что проектные решения, принимаемые при разработке примеров, были основаны на изучении API стандартных объектов Python. Если бы меня попросили свести содержание главы к одной фразе, я бы сказал:

> Создавая объекты в духе Python, наблюдайте за поведением настоящих объектов Python.
>
> – Старинная китайская пословица

Дополнительная литература

В этой главе рассмотрено несколько специальных методов модели данных, поэтому естественно, что ссылки в основном те же, что в главе 1, где был дан общий обзор той же темы. Для удобства я повторю несколько предыдущих рекомендаций и добавлю ряд новых:

Глава «Модель данных» справочного руководства по языку Python (https://docs.python.org/3/reference/datamodel.html)

> Большинство использованных в этой главе методов документированы в разделе 3.3.1 «Простая настройка» (*https://docs.python.org/3/reference/datamodel.html#basic-customization*).

Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», 3-е издание (O'Reilly)

> Глубоко рассмотрены специальные методы.

David Beazley, Brian K. Jones «Python Cookbook», 3-е издание

> В многочисленных рецептах демонстрируются современные подходы к кодированию. Особый интерес представляет глава 8 «Классы и объекты», в которой приведено несколько решений, относящихся к тематике этой главы.

David Beazley «Python Essential Reference», 4-е издание

> Подробно рассматривается модель данных, правда, только в контексте Python 2.6 и Python 3.0 (в четвертом издании). Фундаментальные концепции те же самые, а большинство API модели данных не изменилось со времен Python 2.2, когда были унифицированы встроенные типы и пользовательские классы.

В 2015 году, когда я закончил работу над первым изданием этой книги, Хинек Шлавак (Hynek Schlawack) приступил к разработке пакета `attrs`. Приведу цитату из документации по `attrs`:

> `attrs` – Python-пакет, который вернет **радость** от **написания классов**, освободив вас от рутины реализации объектных протоколов (они же dunder-методы).

Я упоминал `attrs` как более мощную альтернативу `@dataclass` в разделе «Дополнительная литература» главы 5. Построители классов данных, описанные в главе 5, равно как и пакет `attrs`, автоматически наделяют ваши классы несколькими специальными методами. Но умение кодировать эти специальные методы самостоятельно все равно полезно, поскольку позволяет понять, как эти пакеты работают, решить, действительно ли они вам нужны, и – при необходимости – переопределить сгенерированные ими методы.

В этой главе мы рассмотрели все специальные методы, относящиеся к представлению объектов, кроме `__index__` и `__fspath__`. Метод `__index__` мы обсудим в разделе «Метод `__getitem__` с учетом срезов» главы 12. О методе `__fspath__` я говорить не буду. Интересующиеся могут прочитать о нем в документе PEP 519 «Adding a file system path protocol» (*https://peps.python.org/pep-0519/*).

Впервые необходимость различных строковых представлений объекта была осознана в языке Smalltalk. В статье 1996 года «How to Display an Object as a String: printString and displayString» (*http://esug.org/data/HistoricalDocuments/TheSmalltalkReport/ST07/04wo.pdf*) Бобби Вулф обсуждает реализацию методов `printString` и `displayString` в этом языке. Из этой статьи я позаимствовал выражения «в виде, удобном для разработчика» и «в виде, удобном для пользователя» для описания методов `repr()` и `str()` в разделе «Представления объекта» выше.

Поговорим

Свойства позволяют снизить начальные затраты

В первых версиях класса `Vector2d` атрибуты `x` и `y` были открытыми, как и все атрибуты класса и экземпляра по умолчанию. Естественно, пользователям вектора необходим доступ к его компонентам. И хотя наши векторы являются итерируемыми объектами и могут быть распакованы в пару переменных, желательно также иметь возможность писать `my_vector.x` и `my_vector.y` для прямого доступа к компонентам по отдельности.

Осознав необходимость воспрепятствовать случайному изменению атрибутов `x` и `y`, мы реализовали свойства, но больше нигде – ни в коде, ни в открытом интерфейсе класса `Vector2d` – менять ничего не пришлось, что доказывают тесты. Мы по-прежнему можем обращаться к компонентам с помощью нотации `my_vector.x` и `my_vector.y`.

Это доказывает, что начинать разработку класса всегда надо с простейшего варианта, оставив атрибуты открытыми, а когда (и если) мы впоследствии захотим усилить контроль доступа с помощью методов чтения и установки, это можно будет сделать, реализовав свойства и ничего не меняя в уже написанном коде работы с компонентами объекта по именам (например, `x` и `y`), которые первоначально были просто открытыми атрибутами.

Такой подход прямо противоположен пропагандируемому в Java: там программист не может начать с простых атрибутов, а впоследствии, если понадобится, перейти на свойства, потому что таковых в языке попросту не существует. Поэтому написание методов чтения и установки считается нормой в Java – даже если эти методы не делают ничего полезного, – так как при переходе от открытых атрибутов к акцессорам весь ранее написанный код перестанет работать.

Кроме того, как пишут Мартелли, Равенскрофт и Холден в третьем издании книги «Python in a Nutshell», набирать всюду обращения к методам чтения и установки как-то тупо. Приходится писать:

```
>>> my_object.set_foo(my_object.get_foo() + 1)
```

вместо куда более краткого:

```
>>> my_object.foo += 1
```

Уорд Каннингэм, изобретатель вики и основоположник экстремального программирования, рекомендует задавать себе вопрос: «Как написать самый простой код, который будет это делать?» Идея в том, чтобы сосредоточить все внимание на цели[1]. Реализация акцессоров с самого начала только отвлекает от цели. В Python мы можем просто использовать открытые атрибуты, зная, что при необходимости сумеем в любой момент заменить их свойствами.

Закрытые атрибуты – защита и безопасность

> Perl не одержим идеей навязать закрытость во что бы то ни стало. Он предпочитает, чтобы вы не входили в дом, потому что вас туда не приглашали, а не потому, что там стоит пулемет.
> – Ларри Уолл, создатель Perl

Во многих отношениях Python и Perl – полные противоположности, но в вопросе о закрытости объектов Ларри и Гвидо, похоже, едины.

За годы преподавания Python многочисленным программистам на Java я понял, что многие чрезмерно уповают на гарантии закрытости, предоставляемые Java. Но на самом деле модификаторы `private` и `protected` в Java защищают только от непреднамеренных случайностей. Защитить от злого умысла они могут, лишь если приложение развернуто с диспетчером безопасности (*https://docs.oracle.com/javase/ tutorial/essential/environment/security.html*), а такое редко встречается на практике, даже в корпоративной среде.

Для доказательства этого положения я обычно привожу следующий класс Java.

[1] См. «Simplest Thing that Could Possibly Work: A Conversation with Ward Cunningham, Part V» (*http://www.artima.com/intv/simplest3.html*).

Пример 11.20. Confidential.java: класс Java с закрытым полем `secret`

```
public class Confidential {

    private String secret = "";

    public Confidential(String text) {
        this.secret = text.toUpperCase();
    }
}
```

Здесь я сохраняю текст в поле `secret`, предварительно преобразовав его в верхний регистр, чтобы значение этого поля гарантированно было записано заглавными буквами.

Собственно демонстрация заключается в выполнении скрипта *expose.py* интерпретатором Jython. Этот скрипт применяет интроспекцию (в терминологии Java – «отражение»), чтобы получить значение закрытого поля. Код показан в примере 11.21.

Пример 11.21. expose.py: Jython-код для чтения содержимого закрытого поля другого класса

```
#!/usr/bin/env jython
# Примечание: Jython остается на уровне Python 2.7 по состоянию на конец 2020 года
import Confidential
message = Confidential('top secret text')
secret_field = Confidential.getDeclaredField('secret')
secret_field.setAccessible(True) # замок взломан!
print 'message.secret =', secret_field.get(message)
```

Выполнив пример 11.21, получим:

```
$ jython expose.py
message.secret = TOP SECRET TEXT
```

Строка `'TOP SECRET TEXT'` прочитана из закрытого поля `secret` класса `Confidential`.

Никакой черной магии тут нет: скрипт *expose.py* применяет API отражения Java, чтобы получить ссылку на закрытое поле с именем `'secret'`, а затем вызывает метод `'secret_field.setAccessible(True)'`, чтобы сделать его доступным для чтения. Разумеется, то же самое можно сделать и в коде на Java (только придется написать в три раза больше строк, см. файл *Expose.java* в репозитории кода к этой книге по адресу *https://github.com/fluentpython/example-code*).

Решающий вызов `.setAccessible(True)` завершится с ошибкой, только если скрипт Jython или главная программа Java (например, `Expose.class`) работает под управлением диспетчера безопасности SecurityManager. Но на практике Java-приложения редко развертываются таким образом – за исключением Java-апплетов, если вам удастся найти поддерживающий их браузер.

Мой вывод: в Java модификаторы контроля доступа тоже обеспечивают лишь защиту, но не безопасность, по крайней мере на практике. Поэтому расслабьтесь и получайте удовольствие от могущества, которым наделяет вас Python. Но применяйте его ответственно.

Глава 12

Специальные методы для последовательностей

> Не проверяйте, утка ли это; проверяйте, что оно крякает, как утка, ходит, как утка, и т. д. и т. п. – в зависимости от того, какая часть поведения утки важна в ваших языковых игрищах (comp.lang.python, 26 июля 2000).
>
> – Алекс Мартелли

В этой главе мы напишем класс `Vector` для представления многомерного вектора – заметный шаг вперед по сравнению с классом двумерного вектора `Vector2d` из главы 11. Класс `Vector` будет вести себя как стандартная плоская неизменяемая последовательность в Python. Ее элементами будут числа с плавающей точкой, и окончательная версия будет поддерживать следующие возможности:

- базовый протокол последовательности: методы `__len__` и `__getitem__`;
- безопасное представление экземпляров со многими элементами;
- поддержка операции среза, в результате которой получается новый экземпляр `Vector`;
- хеширование агрегата с учетом значений всех содержащихся в нем элементов;
- расширение языка форматирования.

Мы также реализуем доступ к динамическим атрибутам с помощью метода `__getattr__` – как замену доступных только для чтения свойств в классе `Vector2d`, – хотя для типов последовательностей такая функциональность нетипична.

Демонстрация кода будет прерываться обсуждением самой идеи протокола как неформального интерфейса. Мы поговорим о связи между протоколами и *утиной типизацией*, а также о ее практических следствиях для создания пользовательских типов.

Что нового в этой главе

Никаких существенных изменений в этой главе нет. В конец раздела «Протоколы и утиная типизация» добавлен совет, содержащий краткое обсуждение класса `typing.Protocol`.

В разделе «Метод __getitem__ с учетом срезов» реализация `__getitem__` в примере 12.6 сделана более краткой и надежной, чем в первом издании, благодаря

утиной типизации и методу `operator.index`. Это изменение внесено в последующие реализации класса `Vector` в данной главе и в главе 16.

Итак, начнем.

Vector: пользовательский тип последовательности

При реализации класса `Vector` мы будем пользоваться не наследованием, а композицией. Компоненты вектора будут храниться в массиве `array` чисел с плавающей точкой, и мы напишем методы, необходимые для того, чтобы `Vector` вел себя как неизменяемая плоская последовательность.

Но перед тем как приступать к методам последовательностей, разработаем базовую реализацию класса `Vector`, которая будет совместима с написанным ранее классом `Vector2d` – за исключением случаев, где говорить о совместимости не имеет смысла.

Где применяются векторы размерности выше 3

Кому нужен вектор с 1000 измерений? Подсказка: не 3D-дизайнерам! Тем не менее *n*-мерные векторы (с большим значением *n*) широко используются в информационном поиске, где документы и тексты запросов представляются в виде векторов, по одному измерению на каждое слово. Это называется векторной моделью (*http://en.wikipedia.org/wiki/Vector_space_model*). В векторной модели в качестве основной меры релевантности используется коэффициент Отиаи (косинус угла между вектором запроса и вектором документа). При уменьшении угла его косинус стремится к максимальному значению 1, а вместе с ним и релевантность документа запросу.

Однако в этой главе класс `Vector` приведен только в педагогических целях, так что математики почти не будет. У нас более узкая задача – продемонстрировать специальные методы Python в контексте последовательностей.

Для выполнения серьезных математических операций над векторами понадобятся библиотеки NumPy и SciPy. В пакете gemsim (*https://pypi.python.org/pypi/gensim*) Радима Рехурека реализована векторная модель для обработки естественных языков и информационного поиска с использованием NumPy и SciPy.

Vector, попытка № 1: совместимость с Vector2d

Первая версия `Vector` должна быть по возможности совместима с классом `Vector2d`.

Однако же конструктор `Vector` мы не станем делать совместимым. Можно было бы добиться работоспособности выражений `Vector(3, 4)` и `Vector(3, 4, 5)`, разрешив задавать произвольное число аргументов с помощью конструкции `*args` в методе `__init__`, но обычно конструктор последовательности принимает данные в виде итерируемого объекта – как все встроенные типы последовательностей. В примере 12.1 показано несколько способов создания объектов класса `Vector`.

Пример 12.1. Тесты методов `Vector.__init__` и `Vector.__repr__`

```
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

Помимо сигнатуры конструктора, я включил тесты, которые проходили для `Vector2d` (например, `Vector2d(3, 4)`). Они должны проходить и для `Vector` и давать такие же результаты.

Если у вектора больше шести компонент, то вместо окончания строки, порожденной методом `repr()`, выводится ..., как в последней строчке примера 10.1. Это существенно для любого типа коллекции, в котором может быть много элементов, потому что `repr` применяется для отладки (и вряд ли вам понравится, когда один объект занимает тысячи строк на консоли или в журнале). Для создания укороченных представлений используйте модуль `reprlib`, как в примере 12.2. В версии Python 2.7 модуль `reprlib` назывался `repr`.

В примере 12.2 приведена реализация первой версии класса `Vector` (она основана на коде из примеров 11.2 и 11.3).

Пример 12.2. vector_v1.py: основана на vector2d_v1.py

```
from array import array
import reprlib
import math

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components) ❶

    def __iter__(self):
        return iter(self._components) ❷

    def __repr__(self):
        components = reprlib.repr(self._components) ❸
        components = components[components.find('['):-1] ❹
        return f'Vector({components})'

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components)) ❺

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __abs__(self):
```

```
        return math.hypot(*self) ❻

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

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv) ❼
```

❶ В «защищенном» атрибуте экземпляра `self._components` хранится массив `array` компонента `Vector`.
❷ Чтобы было возможно итерирование, возвращаем итератор, построенный по `self._components`[1].
❸ Использовать `reprlib.repr()` для получения представления `self._components` ограниченной длины (например, `array('d', [0.0, 1.0, 2.0, 3.0, 4.0, ...])`).
❹ Удалить префикс `array('d'` и закрывающую скобку `)`, перед тем как подставить строку в вызов конструктора `Vector`.
❺ Построить объект `bytes` из `self._components`.
❻ Начиная с версии Python 3.8 метод `math.hypot` принимает *N*-мерные точки. Раньше я пользовался следующим выражением: `math.sqrt(sum(x * x for x in self))`.
❼ Единственное отличие от написанного ранее метода `frombytes` – последняя строка: мы передаем объект `memoryview` напрямую конструктору, не распаковывая его с помощью `*`, как раньше.

То, как я использовал функцию `reprlib.repr`, заслуживает пояснения. Эта функция порождает безопасное представление длинной или рекурсивной структуры путем ограничения длины выходной строки с заменой отброшенного окончания многоточием `'...'`. Я хотел, чтобы `repr`-представление `Vector` имело вид `Vector([3.0, 4.0, 5.0])`, а не `Vector(array('d', [3.0, 4.0, 5.0]))`, потому что присутствие `array` внутри `Vector` – деталь реализации. Поскольку оба вызова конструктора возвращают одинаковые объекты `Vector`, я предпочел более простой синтаксис с использованием аргумента типа `list`.

При написании метода `__repr__` я мог бы вывести упрощенное отображение `components` с помощью такого выражения: `reprlib.repr(list(self._components))`. Но это было бы расточительно, поскольку пришлось бы копировать каждый элемент `self._components` в `list` только для того, чтобы использовать `list repr`. Вместо этого я решил применить `reprlib.repr` непосредственно к массиву `self._components`, а затем отбросить все символы, оказавшиеся вне квадратных скобок `[]`. Для этого и предназначена вторая строка метода `__repr__` в примере 12.2.

Поскольку метод `repr()` вызывается во время отладки, он никогда не должен возбуждать исключение. Если в `__repr__` происходит какая-то ошибка, вы должны обработать ее сами и сделать все возможное, чтобы показать пользователю нечто разумное, позволяющее идентифицировать получателя (`self`).

[1] Функция `iter()` рассматривается в главе 17 наряду с методом `__iter__`.

Отметим, что методы `__str__`, `__eq__` и `__bool__` остались такими же, как в классе `Vector2d`, а в методе `frombytes` изменился только один символ (удален символ `*` в последней строке). Это воздаяние за то, что класс `Vector2d` изначально был сделан итерируемым.

Кстати, я мог бы сделать `Vector` подклассом `Vector2d`, но не стал по двум причинам. Во-первых, при наличии несовместимых конструкторов создавать подклассы не рекомендуется. Эту трудность можно было бы обойти за счет хитроумной обработки параметров в `__init__`, но есть и вторая, более важная причина: я хочу, чтобы `Vector` был не зависящим от других классов примером реализации протокола последовательности. Этим мы и займемся далее, предварительно обсудив сам термин *протокол*.

Протоколы и утиная типизация

Еще в главе 1 мы видели, что для создания полнофункционального типа последовательности в Python необязательно наследовать какому-то специальному классу; нужно лишь реализовать методы, удовлетворяющие протоколу последовательности. Но что это за протокол такой?

В объектно-ориентированном программировании протоколом называется неформальный интерфейс, определенный только в документации, но не в коде. Например, протокол последовательности в Python подразумевает только наличие методов `__len__` и `__getitem__`. Любой класс `Spam`, в котором есть такие методы со стандартной сигнатурой и семантикой, можно использовать всюду, где ожидается последовательность. Является `Spam` подклассом какого-то другого класса или нет, роли не играет. Мы видели это в примере 1.1, который воспроизведен ниже.

Пример 12.3. Код из примера 1.1, воспроизведенный здесь для удобства

```
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]
```

В классе `FrenchDeck` из примера 12.3 применяются разнообразные средства Python, потому что он реализует протокол последовательности, хотя нигде в коде об этом явно не сказано. Любому опытному программисту на Python достаточно одного взгляда на код, что понять, что это именно класс последо-

вательности, несмотря на то что он является подклассом `object`. Мы говорим, что он *является* последовательностью, потому что *ведет себя* как последовательность, а только это и важно.

Такой подход получил название «утиная типизация», и именно о нем идет речь в высказывании Алекса Мартелли, взятом в качестве эпиграфа к этой главе.

Поскольку протокол – неформальное понятие, которое не подкреплено средствами языка, мы зачастую можем реализовать лишь часть протокола, если точно знаем, в каком контексте будет использоваться класс. Например, для поддержки итерирования нужен только метод `__getitem__`, а без метода `__len__` можно обойтись.

Следуя документу PEP 544 «Protocols: Structural subtyping (static duck typing)» (*https://peps.python.org/pep-0544/*), Python 3.8 поддерживает *классы протоколов*: конструкции из модуля `typing`, которые мы изучали в разделе «Статические протоколы» главы 8. Это новое употребление слова «протокол» в Python имеет родственную, но все же отличающуюся семантику. В тех случаях, когда их необходимо различать, я буду писать *статический протокол*, имея в виду протоколы, формализованные классами протоколов, и *динамический протокол*, имея в виду традиционное понятие. Ключевое различие заключается в том, что реализации статических протоколов должны предоставлять все методы, определенные в классе протокола. Дополнительные сведения см. в разделе «Два вида протоколов» главы 13.

Далее мы реализуем протокол последовательности в классе `Vector`, поначалу без надлежащей поддержки операции среза, но позже добавим и ее.

Vector, попытка № 2: последовательность, допускающая срез

В примере класса `FrenchDeck` мы видели, что поддержать протокол последовательности очень просто, если можно делегировать работу атрибуту объекта, который является последовательностью, в нашем случае таким атрибутом будет массив `self._components`. Для начала нас вполне устроят такие однострочные методы `__len__` и `__getitem__`:

```
class Vector:
    # много строк опущено
    # ...

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

    def __getitem__(self, index):
        return self._components[index]
```

После этих добавлений все показанные ниже операции работают:

```
>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
```

```
>>> v1[0], v1[-1]
(3.0, 5.0)
>>> v7 = Vector(range(7))
>>> v7[1:4]
array('d', [1.0, 2.0, 3.0])
```

Как видите, даже срезы поддерживаются – но не очень хорошо. Было бы лучше, если бы срез вектора также был экземпляром класса `Vector`, а не массивом. В старом классе `FrenchDeck` была такая же проблема: срез оказывался объектом класса `list`. Но в случае `Vector` мы утрачиваем значительную часть функциональности, если операция среза возвращает простой массив.

Рассмотрим встроенные типы последовательностей: для каждого из них операция среза порождает объект того же, а не какого-то другого типа.

Если мы хотим, чтобы срезы `Vector` тоже были объектами класса `Vector`, то не должны делегировать получение среза классу `array`. В методе `__getitem__` мы должны проанализировать полученные аргументы и выполнить подходящее действие.

Теперь посмотрим, как Python преобразует конструкцию `my_seq[1:3]` в аргументы вызова `my_seq.__getitem__(...)`.

Как работает срезка

Код заменяет тысячу слов, поэтому обратимся к примеру 12.4.

Пример 12.4. Изучение поведения `__getitem__` и срезов

```
...     def __getitem__(self, index):
...         return index ❶
...
>>> s = MySeq()
>>> s[1] ❷
1
>>> s[1:4] ❸
slice(1, 4, None)
>>> s[1:4:2] ❹
slice(1, 4, 2)
>>> s[1:4:2, 9] ❺
(slice(1, 4, 2), 9)
>>> s[1:4:2, 7:9] ❻
(slice(1, 4, 2), slice(7,9,None))
```

❶ Здесь `__getitem__` просто возвращает то, что ему передали.
❷ Один индекс, ничего нового.
❸ Нотация `1:4` преобразуется в `slice(1, 4, None)`.
❹ `slice(1, 4, 2)` означает: начать с 1, закончить на 4, с шагом 2.
❺ Сюрприз: при наличии запятых внутри `[]` метод `__getitem` получает кортеж.
❻ Этот кортеж может даже содержать несколько объектов среза.

Теперь приглядимся внимательнее к самому классу `slice`.

Пример 12.5. Инспекция атрибутов класса `slice`

```
>>> slice ❶
<class 'slice'>
>>> dir(slice) ❷
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'indices', 'start', 'step', 'stop']
```

❶ `slice` – встроенный тип (мы это уже поняли в разделе «Объекты среза» главы 2).

❷ Инспекция `slice` показывает наличие атрибутов `start`, `stop` и `step`, а также метода `indices`.

В примере 12.5 вызов `dir(slice)` показывает наличие метода `indices` – весьма интересного, хотя и малоизвестного. Вот что говорит о нем справка – `help(slice.indices)`:

`S.indices(len) -> (start, stop, stride)`

> В предположении, что длина последовательности равна `len`, вычисляет индексы `start` и `stop`, а также длину `stride` расширенного среза, представленного объектом `S`. Индексы, выходящие за границы, приводятся к границам так же, как при обработке обычных срезов.

Иначе говоря, метод `indices` раскрывает нетривиальную логику, применяемую во встроенных последовательностях для корректной обработки отсутствующих или отрицательных индексов и срезов, длина которых превышает длину конечной последовательности. Этот метод возвращает «нормализованные» кортежи, содержащие неотрицательные целые числа `start`, `stop` и `stride`, скорректированные так, чтобы не выходить за границы последовательности заданной длины.

Ниже приведено два примера для последовательности длины 5, например `'ABCDE'`:

```
>>> slice(None, 10, 2).indices(5) ❶
(0, 5, 2)
>>> slice(-3, None, None).indices(5) ❷
(2, 5, 1)
```

❶ `'ABCDE'[:10:2]` – то же самое, что `'ABCDE'[0:5:2]`.

❷ `'ABCDE'[-3]` – то же самое, что `'ABCDE'[2:5:1]`.

В классе `Vector` нам не нужен метод `slice.indices()`, потому что, получив в аргументе срез, мы делегируем его обработку массиву `_components`. Но если опереться на средства, предоставляемые внутренней последовательностью, не получается, то этот метод может сэкономить уйму времени.

Теперь, разобравшись, как обрабатывать срезы, рассмотрим улучшенную реализацию метода `Vector.__getitem__`.

Метод __getitem__ с учетом срезов

В примере 12.6 приведено два метода, необходимых для того, чтобы класс `Vector` вел себя как последовательность: `__len__` и `__getitem__` (последний теперь правильно обрабатывает срезы).

Пример 12.6. Часть файла vector_v2.py: в класс `Vector` из файла vector_v1.py (пример 12.2) добавлены методы `__len__` и `__getitem__`:

```
    def __len__(self):
        return len(self._components)

    def __getitem__(self, key):
        if isinstance(key, slice): ❶
            cls = type(self) ❷
            return cls(self._components[key]) ❸
        index = operator.index(key) ❹
        return self._components[index] ❺
```

❶ Если аргумент `key` принадлежит типу `slice`...
❷ ... то получить класс экземпляра (т. е. `Vector`) и ...
❸ ... вызвать класс для построения нового экземпляра `Vector` по срезу массива `_components`.
❹ Если можно получить `index` по `key`...
❺ ... то просто вернуть один конкретный элемент из `_components`.

Функция `operator.index()` вызывает специальный метод `__index__`. И функция, и специальный метод определены в документе PEP 357 «Allowing Any Object to be Used for Slicing» (*https://peps.python.org/pep-0357/*), в котором Трэвис Олифант предложил разрешить использование в качестве индексов и срезов любого из многочисленных типов целых в NumPy. Основное различие между `operator.index()` и `int()` заключается в том, что первый предназначен для этой конкретной цели. Например, `int(3.14)` возвращает `3`, а `operator.index(3.14)` возбуждает исключение `TypeError`, потому что `float` не может использоваться в качестве индекса.

Злоупотребление функцией `isinstance` иногда является признаком неудачного объектно-ориентированного проектирования, но применение ее для обработки срезов в `__getitem__` оправдано. В первом издании я также применял функцию `isinstance`, чтобы проверить, является ли `key` целым числом. Использование `operator.index` позволяет обойтись без этой проверки и возбудить исключение `TypeError` с очень информативным сообщением, если из `key` нельзя получить `index`. См. последнее сообщение об ошибке в примере 12.7.

После добавления кода из примера 12.6 в класс `Vector` поведение операции среза исправилось, что доказывает пример 12.7.

Пример 12.7. Тесты улучшенного метода `Vector.__getitem__` из примера 12.6

```
>>> v7 = Vector(range(7))
>>> v7[-1] ❶
6.0
>>> v7[1:4] ❷
Vector([1.0, 2.0, 3.0])
>>> v7[-1:] ❸
Vector([6.0])
>>> v7[1,2] ❹
Traceback (most recent call last):
  ...
TypeError: 'tuple' object cannot be interpreted as an integer
```

❶ Если индекс – целое число, то извлекается ровно одна компонента типа `float`.
❷ Если задан индекс типа `slice`, то создается новый объект `Vector`.
❸ Если длина среза `len == 1`, то все равно создается новый объект `Vector`.
❹ Класс `Vector` не поддерживает многомерное индексирование, поэтому при задании кортежа индексов или срезов возбуждается исключение.

Vector, попытка № 3: доступ к динамическим атрибутам

При переходе от класса `Vector2d` к `Vector` мы потеряли возможность обращаться к компонентам вектора по имени, например `v.x`, `v.y`. Теперь мы имеем дело с векторами, имеющими сколь угодно много компонент. Тем не менее иногда удобно обращаться к нескольким первым компонентам по именам, состоящим из одной буквы, например `x`, `y`, `z` вместо `v[0]`, `v[1]` и `v[2]`.

Ниже показан альтернативный синтаксис для чтения первых четырех компонент вектора, который мы хотели бы поддержать:

```
>>> v = Vector(range(10))
>>> v.x
0.0
>>> v.y, v.z, v.t
(1.0, 2.0, 3.0)
```

В классе `Vector2d` мы предоставляли доступ для чтения компонент `x` и `y` с помощью декоратора `@property` (пример 11.7). Мы могли бы завести и в `Vector` четыре свойства, но это утомительно. Специальный метод `__getattr__` позволяет сделать это по-другому и лучше.

Метод `__getattr__` вызывается интерпретатором, если поиск атрибута завершается неудачно. Иначе говоря, анализируя выражение `my_obj.x`, Python проверяет, есть ли у объекта `my_obj` атрибут с именем `x`; если нет, поиск повторяется в классе (`my_obj.__class__`), а затем вверх по иерархии наследования[1]. Если атрибут `x` все равно не найден, то вызывается метод `__getattr__`, определенный в классе `my_obj`, причем ему передается `self` и имя атрибута в виде строки (например, `'x'`).

В примере 12.8 приведен код метода `__getattr__`. Он проверяет, является ли искомый атрибут одной из букв `xyzt`, и если да, то возвращает соответствующую компоненту вектора.

Пример 12.8. Часть файла vector_v3.py: в класс `Vector` из файла vector_v2.py добавлен метод `__getattr__`

```
__match_args__ = ('x', 'y', 'z', 't') ❶

def __getattr__(self, name):
    cls = type(self) ❷
    try:
        pos = cls.__match_args__.index(name) ❸
    except ValueError: ❹
        pos = -1
```

[1] На самом деле поиск атрибутов устроен сложнее. Технические детали мы обсудим в части V, а пока достаточно и этого упрощенного объяснения.

```
    if 0 <= pos < len(self._components): ❺
        return self._components[pos]
    msg = f'{cls.__name__!r} object has no attribute {name!r}' ❻
    raise AttributeError (msg)
```

❶ Инициализировать `__match_args__`, чтобы можно было применить сопоставление с образцом к динамическим атрибутам, поддерживаемым `__getattr__`[1].
❷ Получить класс `Vector` для последующего использования.
❸ Попытаться получить позицию `name` в `__match_args__`.
❹ Вызов `.index(name)` возбуждает исключение `ValueError`, если `name` не найдено; положить `pos` равным –1 (я предпочел бы использовать здесь метод вроде `str.find`, но в классе `tuple` он не реализован).
❺ Если `pos` не выходит за пределы кортежа, вернуть соответствующий элемент.
❻ Если мы дошли до этого места, возбудить исключение `AttributeError` со стандартным сообщением об ошибке.

Реализовать метод `__getattr__` просто, но в данном случае недостаточно. Рассмотрим странное взаимодействие в примере 12.9.

Пример 12.9. Неправильное поведение: присваивание `v.x` не приводит к ошибке, но результат получается несогласованным

```
>>> v = Vector(range(5))
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])
>>> v.x ❶
0.0
>>> v.x = 10 ❷
>>> v.x ❸
3
10
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0]) ❹
```

❶ Доступ к элементу `v[0]` по имени `v.x`.
❷ Присвоить `v.x` новое значение. При этом должно бы возникнуть исключение.
❸ Чтение `v.x` показывает новое значение: 10.
❹ Однако компоненты вектора не изменились.

Сможете объяснить, что здесь происходит? И главное – почему чтение `v.x` возвращает `10`, если это значение не хранится в массиве компонент? Если сходу не понятно, прочитайте еще раз, как работает метод `__getattr__` (перед примером 12.8). Это тонкий момент, но от него зависит многое из того, с чем мы встретимся далее в этой книге.

Поразмыслив над этой проблемой, можете читать дальше – мы объясним, что же случилось.

---

[1] Хотя `__match_args__` введен для поддержки сопоставления с образцами в Python 3.10, установка этого атрибута безвредна и в предшествующих версиях Python. В первом издании книги я назвал его `shortcut_names`. А новое имя позволяет сделать два дела сразу: поддержать позиционные образцы в ветвях `case` и хранить имена динамических атрибутов, поддерживаемые специальной логикой в методах `__getattr__` и `__setattr__`.

Несогласованность в примере 12.9 возникла из-за способа работы `__getattr__`: Python вызывает этот метод только в том случае, когда у объекта нет атрибута с указанным именем. Однако же после присваивания `v.x = 10` у объекта `v` появился атрибут `x`, поэтому `__getattr__` больше не вызывается для доступа к `v.x`: интерпретатор просто вернет значение `10`, связанное с `v.x`. С другой стороны, в нашей реализации `__getattr__` игнорируются все атрибуты экземпляра, кроме `self._components`, откуда читаются значения «виртуальных атрибутов», перечисленных в `__match_args__`.

Чтобы избежать рассогласования, мы должны изменить логику установки атрибутов в классе `Vector`.

Напомним, что в последних вариантах класса `Vector2d` в главе 11 попытка присвоить значение атрибутам экземпляра `.x` или `.y` приводила к исключению `AttributeError`. В классе `Vector` мы хотим возбуждать такое же исключение при любой попытке присвоить значение атрибуту, имя которого состоит из одной строчной буквы, – просто во избежание недоразумений. Для этого реализуем метод `__setattr__`, как показано в примере 12.10.

Пример 12.10. Часть файла vector_v3.py: метод `__setattr__` в классе `Vector`

```
    def __setattr__(self, name, value):
        cls = type(self)
        if len(name) == 1: ❶
            if name in cls.__match_args__: ❷
                error = 'readonly attribute {attr_name!r}'
            elif name.islower(): ❸
                error = "can't set attributes 'a' to 'z' in {cls_name!r}"
            else:
                error = '' ❹
            if error: ❺
                msg = error.format(cls_name=cls.__name__, attr_name=name)
                raise AttributeError(msg)
        super().__setattr__(name,value) ❻
```

❶ Специальная обработка односимвольных имен атрибутов.
❷ Если имя совпадает с одним из перечисленных в `__match_args__`, задать один текст сообщения об ошибке.
❸ Если имя – строчная буква, задать другой текст сообщения – обо всех однобуквенных именах.
❹ В противном случае оставить сообщение об ошибке пустым.
❺ Если сообщение об ошибке не пусто, возбудить исключение.
❻ Случай по умолчанию: вызвать метод `__setattr__` суперкласса для получения стандартного поведения.

Функция `super()` – быстрый способ обратиться к методам суперкласса. Она необходима в динамических языках, поддерживающих множественное наследование, к числу которых относится и Python. Используется для делегирования некоторого действия в подклассе подходящему методу суперкласса, как в примере 12.10. Мы еще вернемся к функции `super` в разделе «Множественное наследование и порядок разрешения методов» главы 14.

Решая, какое сообщение об ошибке вернуть в исключении `AttributeError`, я прежде всего сверился с поведением встроенного типа `complex`, поскольку он неизменяемый и имеет два атрибута: `real` и `imag`. Попытка изменить любой из них приводит к исключению `AttributeError` с сообщением `«can't set attribute»`. С другой стороны, попытка изменить доступный только для чтения атрибут, который защищен, как в разделе «Хешируемый класс `Vector2d`» главы 11, кончается сообщением `«readonly attribute»`. Выбирая значение строки `error` в методе `__setattr__`, я руководствовался обоими образцами, но уточнил, какие именно атрибуты запрещены.

Отметим, что мы не запрещаем установку всех вообще атрибутов, а только таких, имя которых состоит из одной строчной буквы, – чтобы избежать путаницы с доступными только для чтения атрибутами `x`, `y`, `z` и `t`.

Мы знаем, что объявление атрибута `__slots__` на уровне класса предотвращает создание новых атрибутов экземпляров, поэтому может возникнуть искушение воспользоваться этой возможностью и не реализовывать метод `__setattr__`. Но из-за различных подводных камней, которые обсуждались в разделе «Проблемы при использовании __slots__» главы 11, не рекомендуется объявлять `__slots__` только ради запрета создавать новые атрибуты экземпляра. Этот механизм предназначен исключительно для экономии памяти, да и то лишь в случае, когда с этим возникают проблемы.

Но пусть мы и отказались от записи в компоненты `Vector`, все равно из этого примера можно вынести важный урок: часто вместе с методом `__getattr__` приходится писать и метод `__setattr__`, чтобы избежать несогласованного поведения объекта.

Если бы мы решили допустить изменение компонент, то могли бы реализовать метод `__setitem__`, чтобы можно было писать `v[0] = 1.1`, и (или) метод `__setattr__`, чтобы работала конструкция `v.x = 1.1`. Но сам класс `Vector` должен оставаться неизменяемым, потому что в следующем разделе мы собираемся сделать его хешируемым.

Vector, попытка № 4: хеширование и ускорение оператора ==

И снова нам предстоит реализовать метод `__hash__`. В сочетании с уже имеющимся методом `__eq__` это сделает экземпляры класса `Vector` хешируемыми.

Метод `__hash__` в классе `Vector2d` (пример 11.8) вычислял хеш кортежа с двумя элементами: `self.x` и `self.y`. Теперь же элементов может быть тысячи, поэтому построение кортежа обошлось бы слишком дорого. Вместо этого я применю оператор `^` (ИСКЛЮЧАЮЩЕЕ ИЛИ) к хешам всех компонент, например `v[0] ^ v[1] ^ v[2]`. Тут нам на помощь придет функция `functools.reduce`. Выше я говорил, что функция `reduce` уже не так популярна, как в былые времена[1], но для вычисления хеша всех компонент она подходит идеально. На рис. 12.1 представлена общая идея функции `reduce`.

[1] Функции `sum`, `any` и `all` покрывают большинство типичных применений `reduce`. См. обсуждение в разделе «Современные замены `map`, `filter` и `reduce`» главы 7.

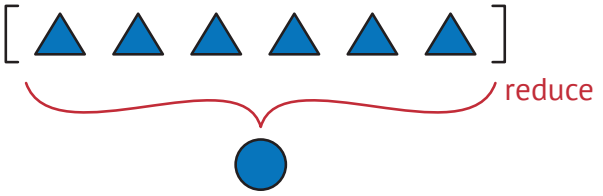


Рис. 12.1. Редуцирующие функции – `reduce`, `sum`, `any`, `all` – порождают единственное значение-агрегат из последовательности или произвольного конечного итерируемого объекта

До сих пор мы видели, что функцию `functools.reduce()` можно заменить функцией `sum()`, а теперь объясним, как же она все-таки работает. Идея в том, чтобы редуцировать последовательность значений в единственное значение. Первый аргумент `reduce()` – функция с двумя аргументами, а второй – итерируемый объект. Допустим, что имеется функция с двумя аргументами `fn` и список `lst`. Если написать `reduce(fn, lst)`, то `fn` сначала применяется к первым двум элементам – `fn(lst[0], lst[1])`, – и в результате получится первый результат `r1`. Затем `fn` применяется к `r1` и следующему элементу – `fn(r1, lst[2])`; так мы получаем второй результат `r2`. Потом вызов `fn(r2, lst[3])` порождает `r3` ... и так далее до последнего элемента, после чего возвращается окончательный результат `rN`.

Вот как можно было бы применить `reduce` для вычисления 5! (факториал 5):

```
>>> 2 * 3 * 4 * 5 # ожидаемый результат: 5! == 120
120
>>> import functools
>>> functools.reduce(lambda a,b: a*b, range(1, 6))
120
```

Но вернемся к проблеме хеширования. В примере 12.11 показано, как можно было бы вычислить результат многократного применения `^` тремя способами: один – с помощью цикла `for` и два – с помощью `reduce`.

Пример 12.11. Три способа вычислить результат применения оператора ИСКЛЮЧАЮЩЕЕ ИЛИ к целым числам от 0 до 5

```
>>> n = 0
>>> for i in range(1, 6): ❶
...     n ^= i
...
>>> n
1
>>> import functools
>>> functools.reduce(lambda a, b: a^b, range(6)) ❷
1
>>> import operator
>>> functools.reduce(operator.xor, range(6)) ❸
1
```

❶ Агрегирование в цикле `for` в накопительную переменную.

❷ `functools.reduce` с анонимной функцией.

❸ `functools.reduce` с заменой специально написанного лямбда-выражения функцией `operator.xor`.

Из представленных вариантов мне больше всего нравится последний, а на втором месте стоит цикл `for`. А вам как кажется?

В разделе «Модуль operator» главы 7 мы видели, что модуль `operator` предоставляет функциональность всех инфиксных операторов Python в форме функций, снижая потребность в лямбда-выражениях.

Чтобы написать метод `Vector.__hash__` в том стиле, который я предпочитаю, необходимо импортировать модули `functools` и `operator`. Изменения показаны в примере 12.12.

Пример 12.12. Часть файла vector_v4.py: в класс `Vector` из файла vector_v3.py добавлены два предложения импорта и метод `__hash__`

```
from array import array
import reprlib
import math
import functools ❶
import operator ❷

class Vector:
    typecode = 'd'

    # много строк опущено...

    def __eq__(self, other): ❸
        return tuple(self) == tuple(other)

    def __hash__(self):
        hashes = (hash(x) for x in self._components) ❹
        return functools.reduce(operator.xor, hashes, 0) ❺

    # последующие строки опущены...
```

❶ Импортировать `functools` для использования `reduce`.
❷ Импортировать `operator` для использования `xor`.
❸ Метод `__eq__` не изменился; я привел его только потому, что методы `__eq__` и `__hash__` принято располагать в исходном коде рядом, т. к. они дополняют друг друга.
❹ Создать генераторное выражение для отложенного вычисления хеша каждой компоненты.
❺ Подать выражение `hashes` на вход `reduce` вместе с функцией `xor` – для вычисления итогового хеш-значения; третий аргумент, равный 0, – инициализатор (см. предупреждение ниже).

При использовании `reduce` рекомендуется задавать третий аргумент, `reduce(function, iterable, initializer)`, чтобы предотвратить появление исключения `TypeError: reduce() of empty sequence with no initial value` (отличное сообщение: описывается проблема и способ исправления). Значение `initializer` возвращается, если последовательность пуста, а кроме того, используется в качестве первого аргумента в цикле редукции, поэтому оно должно быть нейтральным элементом относительно выполняемой операции. Так, для операций `+`, `|`, `^` `initializer` должен быть равен `0`, а для `*`, `&` – `1`.

Метод `__hash__` в примере 11.12 – отличный пример техники map-reduce (рис. 12.2).

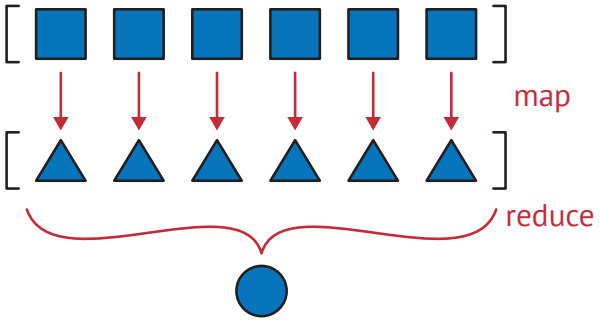


Рис. 12.2. Map-reduce: применить функцию к каждому элементу для генерации новой последовательности (map), затем вычислить агрегат (reduce)

На шаге отображения (map) порождается один хеш для каждого компонента, а на шаге редукции (reduce) все хеши агрегируются с помощью оператора `xor`. Если использовать функцию `map` вместо генераторного выражения, то шаг отображения станет даже более наглядным:

```
def __hash__(self):
    hashes = map(hash, self._components)
    return functools.reduce(operator.xor, hashes)
```

Решение на основе `map` не так эффективно в Python 2, где функция `map` строит список, содержащий результаты. Однако в Python 3 `map` откладывает вычисления: она порождает генератор, который отдает результаты по требованию, экономя тем самым память, – точно так же, как генераторное выражение в методе `__hash__` из примера 12.8.

Раз уж мы заговорили о редуцирующих функциях, то почему бы не заменить нашу написанную на скорую руку реализацию оператора `__eq__` другой, которая и работать будет быстрее, и памяти потреблять меньше, по крайней мере для больших векторов. В примере 11.2 приведена такая лаконичная реализация `__eq__`:

```
def __eq__(self, other):
    return tuple(self) == tuple(other)
```

Она работает для `Vector2d` и для `Vector` – и даже считает, что `Vector([1, 2])` равен `(1, 2)`; это может оказаться проблемой, однако пока закроем на нее глаза[1]. Но для векторов с тысячами компонент эта реализация крайне неэффективна. Она строит два кортежа, полностью копируя оба операнда, только для того, чтобы воспользоваться оператором `__eq__` из типа `tuple`. Такая экономия усилий вполне оправдана для класса `Vector2d` (всего с двумя компонентами), но

[1] К вопросу о разумности равенства `Vector([1, 2]) == (1, 2)` мы серьезно подойдем в разделе «Основы перегрузки операторов» главы 16.

не для многомерных векторов. Более эффективный способ сравнения объекта `Vector` с другим объектом `Vector` или с итерируемым объектом показан в примере 12.13.

Пример 12.13. Метод `Vector.eq`, в котором используется функция `zip` в цикле `for` для более эффективного сравнения

```
    def __eq__(self, other):
        if len(self) != len(other): ❶
            return False
        for a, b in zip(self, other): ❷
            if a != b: ❸
                return False
        return True ❹
```

❶ Если длины объектов различны, то они не равны.
❷ Функция `zip` порождает генератор кортежей, содержащих соответственные элементы каждого переданного ей итерируемого объекта. Если вы с ней незнакомы, см. врезку «Удивительная функция zip» ниже. Сравнение длин в предложении ❶ необходимо, потому что `zip` без предупреждения перестает порождать значения, как только хотя бы один входной аргумент оказывается исчерпанным.
❸ Как только встречаются две различные компоненты, выходим и возвращаем `False`.
❹ В противном случае объекты равны.

Свое название функция `zip` получила от застежки-молнии (zipper), принцип работы которой основан на зацеплении зубьев, расположенных с двух сторон, – наглядная аналогия того, что происходит при обращении к `zip(left, right)`. Никакого отношения к формату сжатия файлов это название не имеет.

Код из примера 12.13 эффективен, но функция `all` может вычислить тот же агрегат, что и цикл `for`, всего в одной строчке: если все сравнения соответственных компонент операндов возвращают `True`, то и результат равен `True`. Как только какое-нибудь сравнение возвращает `False`, так `all` сразу возвращает `False`. В примере 12.14 показано, как выглядит метод `__eq__`, в котором используется `all`.

Пример 12.14. Оператор `Vector.eq` с использованием `zip` и `all`: логика та же, что в примере 12.13

```
    def __eq__(self, other):
        return len(self) == len(other) and all(a == b for a, b in zip(self, other))
```

Отметим, что сначала проверяется равенство длин операндов, потому что `zip` остановится по исчерпании более короткого операнда.

Реализацию из примера 12.14 мы включили в файл *vector_v4.py*.

Удивительная функция zip

Наличие цикла `for`, в котором можно обойти элементы коллекции без возни с индексной переменной, – отличное дело, и многие ошибки так можно предотвратить, только для этого нужны специальные служебные функции. Одна из них – встроенная функция `zip`, позволяющая параллельно обходить два и более итерируемых объекта: она возвращает кортежи, которые можно распаковать в переменные, – по одной для каждого входного объекта. См. пример 12.15.

Пример 12.15. Встроенная функция `zip` за работой

```
>>> zip(range(3), 'ABC') ❶
<zip object at 0x10063ae48>
>>> list(zip(range(3), 'ABC')) ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(zip(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3])) ❸
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2)]
>>> from itertools import zip_longest ❹
>>> list(zip_longest(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3],
fillvalue=-1))
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2), (-1, -1, 3.3)]
```

❶ `zip` возвращает генератор, который порождает кортежи по запросу.

❷ Здесь мы строим из генератора список `list` просто для отображения; обычно генератор обходят в цикле.

❸ `zip` останавливается, не выдавая предупреждения, как только один из итерируемых объектов оказывается исчерпанным.

❹ Функция `itertools.zip_longest` ведет себя иначе: она подставляет вместо отсутствующих значений необязательный аргумент `fillvalue` (по умолчанию `None`), поэтому генерирует кортежи, пока не окажется исчерпанным самый длинный итерируемый объект.

Новый параметр zip() в Python 3.10

В первом издании книги я писал, что поведение `zip` – остановка без предупреждения по исчерпании самого короткого итерируемого объекта – кажется мне неожиданным, а для API это не есть хорошо. Молчаливое игнорирование части входных данных может приводить к тонким ошибкам. Вместо этого `zip` должна была бы возбуждать исключение `ValueError`, если не все итерируемые объекты имеют одинаковую длину, что и имеет место при распаковке итерируемого объекта в кортеж переменных разной длины, – в полном соответствии с принципом быстрого отказа, принятым в Python. В документе PEP 618 «Add Optional Length-Checking To zip» (*https://peps.python.org/pep-0618/*) предложено добавить в `zip` факультативный аргумент `strict`, при наличии которого она ведет себя именно таким образом. Предложение реализовано в Python 3.10.

Функцию `zip` можно также использовать для транспонирования матрицы, представленной в виде вложенных итерируемых объектов. Например:

```
>>> a = [(1, 2, 3),
...      (4, 5, 6)]
>>> list(zip(*a))
[(1, 4), (2, 5), (3, 6)]
>>> b = [(1, 2),
...      (3, 4),
...      (5, 6)]
>>> list(zip(*b))
[(1, 3, 5), (2, 4, 6)]
```

Если вы хотите до конца разобраться, как работает `zip`, потратьте некоторое время на анализ этих примеров.

Встроенная функция `enumerate` – еще одна генераторная функция, которую часто используют в циклах `for`, чтобы избежать явной работы с индексными переменными. Если вы незнакомы с `enumerate`, обязательно прочитайте раздел документации «Встроенные функции» (*https://docs.python.org/3/library/functions.html#enumerate*). Функции `zip`, `enumerate` и другие генераторные функции из стандартной библиотеки рассматриваются в разделе «Генераторные функции в стандартной библиотеке» главы 17.

И в завершение этой главы перенесем метод `__format__` из класса `Vector2d` в класс `Vector`.

Vector, попытка № 5: форматирование

Метод `__format__` класса `Vector` будет похож на одноименный метод из класса `Vector2d`, но вместо специального представления в полярных координатах мы будем использовать так называемые «гиперсферические» координаты (название связано с тем, что в пространствах размерности 4 и выше сферы называются гиперсферами)[1]. Соответственно, специальный суффикс форматной строки `'p'` мы заменим на `'h'`.

В разделе «Форматированное отображение» главы 11 мы видели, что при расширении мини-языка спецификации формата (*https://docs.python.org/3/library/string.html#formatspec*) лучше не использовать форматные коды, предназначенные для встроенных типов. В частности, в нашем расширенном мини-языке коды `'eEfFgGn%'` используются в своем изначальном смысле, поэтому их-то точно нельзя переопределять. Для форматирования целых чисел служат коды `'bcdoxXn'`, а для строк – код `'s'`. Для вывода объекта `Vector2d` в полярных координатах я выбрал код `'p'`, а для гиперсферических координат возьму код `'h'`.

Например, для объекта `Vector` в четырехмерном пространстве `(len(v) == 4)` код `'h'` порождает представление вида `<r, Φ1, Φ2, Φ3>`, где `r` – модуль вектора (`abs(v)`), а `Φ1, Φ2, Φ3` – угловые координаты.

[1] На сайте Wolfram Mathworld (*http://mathworld.wolfram.com/Hypersphere.html*) имеется статья о гиперсферах; в Википедии запрос по слову «hypersphere» переадресуется на статью «n-sphere» (*http://en.wikipedia.org/wiki/Nsphere*).

Ниже приведены примеры вывода 4-мерного вектора в сферических координатах, взятые из тестов в файле *vector_v5.py* (см. пример 12.16):

```
>>> format(Vector([-1, -1, -1, -1]), 'h')
'<2.0, 2.0943951023931957, 2.186276035465284,
3.9269908169872414>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
```

Прежде чем вносить мелкие изменения в метод `__format__`, мы должны написать два вспомогательных метода: `angle(n)` будет вычислять одну из угловых координат (например, ϕ_1), а `angles()` – возвращать итерируемый объект, содержащий все угловые координаты. Не стану останавливаться здесь на математической теории, интересующиеся читатели могут найти формулы преобразования из декартовых координат в сферические в статье из Википедии (*http://en.wikipedia.org/wiki/N-sphere*).

В примере 12.16 приведен полный код из файла *vector_v5.py*, в который вошло все, что мы сделали, начиная с раздела «Vector, попытка № 1: совместимость с Vector2d», включая форматирование.

Пример 12.16. vector_v5.py: тесты и окончательный код класса `Vector`; выноски описывают добавления, необходимые для поддержки метода `__format__`

```
"""
Многомерный класс ``Vector``, попытка 5

A ``Vector`` is built from an iterable of numbers::

    >>> Vector([3.1, 4.2])
    Vector([3.1, 4.2])
    >>> Vector((3, 4, 5))
    Vector([3.0, 4.0, 5.0])
    >>> Vector(range(10))
    Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])


Tests with two dimensions (same results as ``vector2d_v1.py``)::

    >>> v1 = Vector([3, 4])
    >>> x, y = v1
    >>> x, y
    (3.0, 4.0)
    >>> v1
    Vector([3.0, 4.0])
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0)
    >>> octets = bytes(v1)
    >>> octets

  b'd||x00||x00||x00||x00||x00||x00||x08@||x00||x00||x00||x00||x00||x00||x10@'
```

```
    >>> abs(v1)
    5.0
    >>> bool(v1), bool(Vector([0, 0]))
    (True, False)


Test of ``.frombytes()`` class method:

    >>> v1_clone = Vector.frombytes(bytes(v1))
    >>> v1_clone
    Vector([3.0, 4.0])
    >>> v1 == v1_clone
    True


Tests with three dimensions::

    >>> v1 = Vector([3, 4, 5])
    >>> x, y, z = v1
    >>> x, y, z
    (3.0, 4.0, 5.0)
    >>> v1
    Vector([3.0, 4.0, 5.0])
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0, 5.0)
    >>> abs(v1) # doctest:+ELLIPSIS
    7.071067811...
    >>> bool(v1), bool(Vector([0, 0, 0]))
    (True, False)


Tests with many dimensions::

    >>> v7 = Vector(range(7))
    >>> v7
    Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
    >>> abs(v7) # doctest:+ELLIPSIS
    9.53939201...


Test of ``.__bytes__`` and ``.frombytes()`` methods::

    >>> v1 = Vector([3, 4, 5])
    >>> v1_clone = Vector.frombytes(bytes(v1))
    >>> v1_clone
    Vector([3.0, 4.0, 5.0])
    >>> v1 == v1_clone
    True


Tests of sequence behavior::

    >>> v1 = Vector([3, 4, 5])
```

```
    >>> len(v1)
    3
    >>> v1[0], v1[len(v1)-1], v1[-1]
    (3.0, 5.0, 5.0)


Test of slicing::

    >>> v7 = Vector(range(7))
    >>> v7[-1]
    6.0
    >>> v7[1:4]
    Vector([1.0, 2.0, 3.0])
    >>> v7[-1:]
    Vector([6.0])
    >>> v7[1,2]
    Traceback (most recent call last):
      ...
    TypeError: 'tuple' object cannot be interpreted as an integer


Tests of dynamic attribute access::

    >>> v7 = Vector(range(10))
    >>> v7.x
    0.0
    >>> v7.y, v7.z, v7.t
    (1.0, 2.0, 3.0)

Dynamic attribute lookup failures::

    >>> v7.k
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 'k'
    >>> v3 = Vector(range(3))
    >>> v3.t
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 't'
    >>> v3.spam
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 'spam'


Tests of hashing::

    >>> v1 = Vector([3, 4])
    >>> v2 = Vector([3.1, 4.2])
    >>> v3 = Vector([3, 4, 5])
    >>> v6 = Vector(range(6))
    >>> hash(v1), hash(v3), hash(v6)
    (7, 2, 1)
```

```
Most hash values of non-integers vary from a 32-bit to 64-bit CPython build::

    >>> import sys
    >>> hash(v2) == (384307168202284039 if sys.maxsize > 2**32 else 357915986)
    True

Tests of ``format()`` with Cartesian coordinates in 2D::

    >>> v1 = Vector([3, 4])
    >>> format(v1)
    '(3.0, 4.0)'
    >>> format(v1, '.2f')
    '(3.00, 4.00)'
    >>> format(v1, '.3e')
    '(3.000e+00, 4.000e+00)'

Tests of ``format()`` with Cartesian coordinates in 3D and 7D::

    >>> v3 = Vector([3, 4, 5])
    >>> format(v3)
    '(3.0, 4.0, 5.0)'
    >>> format(Vector(range(7)))
    '(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)'

Tests of ``format()`` with spherical coordinates in 2D, 3D and 4D::

    >>> format(Vector([1, 1]), 'h') # doctest:+ELLIPSIS
    '<1.414213..., 0.785398...>'
    >>> format(Vector([1, 1]), '.3eh')
    '<1.414e+00, 7.854e-01>'
    >>> format(Vector([1, 1]), '0.5fh')
    '<1.41421, 0.78540>'
    >>> format(Vector([1, 1, 1]), 'h') # doctest:+ELLIPSIS
    '<1.73205..., 0.95531..., 0.78539...>'
    >>> format(Vector([2, 2, 2]), '.3eh')
    '<3.464e+00, 9.553e-01, 7.854e-01>'
    >>> format(Vector([0, 0, 0]), '0.5fh')
    '<0.00000, 0.00000, 0.00000>'
    >>> format(Vector([-1, -1, -1, -1]), 'h') # doctest:+ELLIPSIS
    '<2.0, 2.09439..., 2.18627..., 3.92699...>'
    >>> format(Vector([2, 2, 2, 2]), '.3eh')
    '<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
    >>> format(Vector([0, 1, 0, 0]), '0.5fh')
    '<1.00000, 1.57080, 0.00000, 0.00000>'
"""

from array import array
import reprlib
import math
import functools
import operator
import itertools ❶

class Vector:
    typecode = 'd'
```

```
    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        return iter(self._components)

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[components.find('['):-1]
        return f'Vector({components})'

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components))

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))

    def __hash__(self):
        hashes = (hash(x) for x in self)
        return functools.reduce(operator.xor, hashes, 0)

    def __abs__(self):
        return math.hypot(*self)

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

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

    def __getitem__(self, key):
        if isinstance(key, slice):
            cls = type(self)
            return cls(self._components[key])
        index = operator.index(key)
        return self._components[index]

    __match_args__ = ('x', 'y', 'z', 't')

    def __getattr__(self, name):
        cls = type(self)
        try:
            pos = cls.__match_args__.index(name)
        except ValueError:
            pos = -1
        if 0 <= pos < len(self._components):
            return self._components[pos]
        msg = f'{cls.__name__!r} object has no attribute {name!r}'
        raise AttributeError(msg)
```

```
    def angle(self, n): ❷
        r = math.hypot(*self[n:])
        a = math.atan2(r, self[n-1])
        if (n == len(self) - 1) and (self[-1] < 0):
            return math.pi * 2 - a
        else:
            return a

    def angles(self): ❸
        return (self.angle(n) for n in range(1, len(self)))

    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('h'): # гиперсферические координаты
            fmt_spec = fmt_spec[:-1]
            coords = itertools.chain([abs(self)],
                                     self.angles()) ❹
            outer_fmt = '<{}>' ❺
        else:
            coords = self
            outer_fmt = '({})' ❻
        components = (format(c, fmt_spec) for c in coords) ❼
        return outer_fmt.format(', '.join(components)) ❽

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv)
```

❶ Импортировать `itertools`, чтобы можно было воспользоваться функцией `chain` в методе `__format__`.
❷ Вычислить одну из угловых координат по формулам, взятым из статьи об *N-сфере* (*http://en.wikipedia.org/wiki/N-sphere*).
❸ Создать генераторное выражение для вычисления всех угловых координат по запросу.
❹ Использовать `itertools.chain` для порождения генераторного выражения, которое перебирает модуль и угловые координаты вектора.
❺ Сконфигурировать отображение сферических координат в угловых скобках.
❻ Сконфигурировать отображение декартовых координат в круглых скобках.
❼ Создать генераторное выражение для форматирования координат по запросу.
❽ Подставить отформатированные компоненты, разделенные запятыми, в угловые или круглые скобки.

Мы вовсю пользуемся генераторными выражениями в методах `__format__`, `angle` и `angles`, но наша цель здесь – просто написать метод `__format__`, чтобы класс `Vector` не уступал в полноте реализации классу `Vector2d`. При рассмотрении генераторов в главе 14 мы воспользуемся в качестве примера кодом из класса `Vector`, и тогда все хитрости будут подробно объяснены.

Итак, все задачи, которые мы ставили перед собой в этой главе, решены. В главе 16 мы пополним класс `Vector` инфиксными операторами, но пока хотели лишь изучить, как писать специальные методы, полезные в различных классах коллекций.

Резюме

Класс `Vector` из этой главы был задуман совместимым с классом `Vector2d` во всем, кроме использования другой сигнатуры конструктора, – теперь он принимает один итерируемый объект, как конструкторы встроенных типов последовательностей. Чтобы класс `Vector` вел себя так же, как последовательность, оказалось достаточно реализовать методы `__getitem__` и `__len__`, и этот факт подвиг нас на обсуждение протоколов – неформальных интерфейсов в языках с утиной типизацией.

Далее мы разобрались, как на самом деле работает конструкция `my_seq[a:b:c]`, для чего создали объект `slice(a, b, c)` и передали его методу `__getitem__`. Вооружившись этими знаниями, мы переделали класс `Vector`, так чтобы операция получения среза выполнялась для него корректно, т. е. возвращала экземпляр типа `Vector`, как и положено последовательности в Python.

Нашим следующим шагом было обеспечение доступа для чтения к нескольким первым компонентам объекта `Vector` по именам, т. е. с помощью нотации `my_vec.x`. Для этого мы реализовали метод `__getattr__`. При этом у пользователя могло возникнуть искушение присвоить значение таким компонентам, написав `my_vec.x = 7`, однако это приводило к ошибке. Мы исправили ошибку, реализовав еще и метод `__setattr__`, запрещающий присваивать значения атрибутам с однобуквенными именами. Очень часто бывает, что методы `__getattr__` и `__setattr__` необходимо реализовывать совместно во избежание несогласованного поведения.

Реализация метода `__hash__` предоставила нам отличную возможность воспользоваться функцией `functools.reduce`, поскольку необходимо было применить оператор `^` к хешам всех компонент `Vector`, чтобы создать агрегированное хеш-значение объекта `Vector` в целом. Применив функцию `reduce` в методе `__hash__`, мы затем воспользовались встроенной функцией `all` для создания более эффективной версии метода `__eq__`.

И последним усовершенствованием класса `Vector` стала новая реализация метода `__format__` из класса `Vector2d`, поддерживающая гиперсферические координаты в дополнение к декартовым. Тут нам понадобились кое-какие математические формулы и несколько генераторов, но все это детали реализации (к генераторам мы еще вернемся в главе 17). В последнем разделе нашей целью было поддержать специальный формат и тем самым выполнить данное ранее обещание, что класс `Vector` сможет делать все, что умел `Vector2d`, и кое-что сверх того.

Как и в главе 11, мы часто оглядывались на поведение стандартных объектов Python, стремясь имитировать его, чтобы класс `Vector` соответствовал духу Python.

В главе 16 мы реализуем в классе `Vector` несколько инфиксных операторов. Математика будет куда проще, чем в методе `angle()`, зато изучение работы инфиксных операторов в Python станет отличным уроком по объектно-ориентированному проектированию. Однако прежде чем заняться перегрузкой операторов, мы на время отвлечемся от разработки отдельного класса и посмотрим, как можно организовать несколько классов с помощью интерфейсов и наследования. Это темы глав 13 и 14.

Дополнительная литература

Большинство специальных методов, рассмотренных при разработке класса `Vector`, встречаются и в классе `Vector2d` из главы 11, поэтому актуальны все библиографические ссылки, приведенные в предыдущей главе.

Мощную функцию высшего порядка `reduce` называют также fold, accumulate, aggregate, compress и inject. Дополнительные сведения можно найти в статье из Википедии «Fold (higher-order function)» (*http://en.wikipedia.org/wiki/Fold_(higher-order_function)*), где описаны применения этой функции с упором на функциональное программирование с рекурсивными структурами данных. В этой статье имеется также таблица, в которой перечислены похожие функции в десятках языков программирования.

В документе «What's New in Python 2.5» (*https://docs.python.org/2.5/whatsnew/pep-357.html*) имеется короткое объяснение метода `__index__`, предназначенного для поддержки методов `__getitem__`, как было показано в разделе «Метод __getitem__ с учетом срезов». В документе PEP 357 «Allowing Any Object to be Used for Slicing» (*https://peps.python.org/pep-0357/*) обосновывается его необходимость с точки зрения разработчика C-расширения, Трэвиса Олифанта, основного автора NumPy. Благодаря многочисленным вкладам Олифанта Python вышел на первое место среди языков программирования для научной работы, а затем и на лидирующие позиции в области приложений машинного обучения.

Поговорим

Протоколы как неформальные интерфейсы

Протоколы – не изобретение Python. Авторы языка Smalltalk, пустившие в оборот также выражение «объектно-ориентированный», использовали слово «протокол» как синоним того, что сейчас называется интерфейсом. В некоторых средах программирования на Smalltalk программистам разрешалось помечать группу методов как протокол, но только в целях документирования и навигации – сам язык эту концепцию не поддерживал. Поэтому я полагаю, что «неформальный интерфейс» – разумное краткое объяснение существа «протокола» в выступлении перед аудиторией, больше знакомой с формальными (и поддержанными компилятором) интерфейсами.

Протоколы естественно возникают в любом языке с динамической типизацией, когда контроль типов производится во время выполнения, потому что в объявлениях переменных и сигнатур методов нет никакой статической информации о типе. Ruby – еще один важный объектно-ориентированный язык, в котором имеется динамическая типизация и используются протоколы.

В документации по Python протокол можно узнать по выражениям типа «объект, похожий на файл». Это сокращение фразы «нечто, что ведет себя в достаточной степени похоже на файл благодаря реализации тех частей интерфейса файла, которые существенны в данном контексте».

Кто-то решит, что реализация лишь части протокола – признак небрежного программирования, но у такого подхода есть преимущество – простота. В разделе 3.3 главы «Модель данных» (*https://docs.python.org/3/reference/datamodel.html# special-method-names*) читаем такую рекомендацию:

> При реализации класса, имитирующего встроенный тип, важно не заходить слишком далеко, а ограничиться лишь тем, что имеет смысл для моделируемого объекта. Например, для некоторых последовательностей вполне достаточно извлечения отдельных элементов, тогда как получение среза бессмысленно.

Если мы можем обойтись без кодирования ненужных методов только для того, чтобы удовлетворить требованиям какого-то перенасыщенного функциональностью контракта, а компилятор при этом не будет ругаться, то становится проще следовать принципу KISS[1] (*http://en.wikipedia.org/wiki/KISS_principle*).

С другой стороны, если мы пользуемся средством проверки типов для верификации реализаций протоколов, то необходимо более строгое определение протокола. Именно такое определение и предлагает класс `typing.Protocol`.

Мы еще вернемся к протоколам и интерфейсам в главе 13, которая в основном этой теме и посвящена.

Истоки утиной типизации

Я полагаю, что популяризации термина «duck typing» (утиная типизация) больше других способствовало сообщество Ruby, обращавшееся с проповедью к поклонникам Java. Но это выражение встречалось в обсуждениях Python еще до того, как Ruby и Python стали «популярными». Согласно Википедии, один из первых примеров аналогии с уткой в объектно-ориентированном программировании – сообщение в списке рассылки Python, отправленное Алексом Мартелли 26 июля 2000 года и касавшееся полиморфизма (с заголовком Re: Type checking in python?) (*https://mail.python.org/pipermail/python-list/2000-July/046184.html*). Именно из него взята цитата, ставшая эпиграфом к данной главе. Если вам любопытны литературные корни термина «утиная типизация», а также применения этой объектно-ориентированной концепции во многих языках, почитайте статью Википедии «Duck typing» (*http://en.wikipedia.org/wiki/Duck_typing*).

Безопасный __format__ повышенного удобства

При реализации метода `__format__` мы не принимали никаких мер предосторожности на случай экземпляров `Vector` с очень большим числом компонент, хотя в методе `__repr__` применили для этой цели библиотеку `reprlib`. Обоснованием служит тот факт, что функция `repr()` предназначена для отладки и протоколирования, поэтому любой ценой должна вывести хоть какое-то полезное представление, тогда как `__format__` предназначен для конечного пользователя, который, вероятно, хочет видеть вектор целиком. Если вы полагаете, что это опасно, то можете продолжить расширение мини-языка спецификации формата.

Я бы сделал это так: по умолчанию для любого форматированного вектора выводится разумное, хотя и ограниченное количество компонент, скажем 30. Если элементов больше, то поведение по умолчанию может быть аналогично тому, что делает `reprlib`: отбросить дополнительные компоненты, заменив их многоточием. Но если спецификатор формата заканчивается специальным кодом `*`, означающим «все», то ограничения на размер не действуют. Таким образом, пользователь, который не знает о проблеме очень длинного представления, не станет жертвой случайности. Однако если ограничение начинает мешать, то наличие … должно натолкнуть пользователя на мысль поискать в документации, где он узнает о коде форматирования `*`.

[1] Keep it simple, stupid – Будь проще, глупышка. – *Прим. перев.*

Как вычислить сумму в духе Python

Не существует однозначного ответа на вопрос «Что соответствует духу Python?», как и ответа на вопрос «Что такое красота?». Я часто говорю, что это означает «идиоматичный Python», однако такой ответ не вполне годится, потому что слово «идиоматичный» для меня и для вас может означать разные вещи. Но одно я знаю точно: «идиоматичность» не означает, что нужно использовать средства языка, спрятанные в самых потаенных закоулках.

В списке рассылки Python (*https://mail.python.org/mailman/listinfo/python-list*) есть ветка, датированная апрелем 2003, под названием «Pythonic Way to Sum n-th List Element?» (*https://mail.python.org/pipermail/python-list/2003-April/218568.html*). Она примыкает к обсуждению функции `reduce` в этой главе.

Начавший ее Гай Миддлтон просил улучшить следующее решение, оговорившись, что ему не нравятся лямбда-выражения[1]:

```
>>> my_list = [[1, 2, 3], [40, 50, 60], [9, 8, 7]]
>>> import functools
>>> functools.reduce(lambda a, b: a+b, [sub[1] for sub in my_list])
60
```

В этом коде идиом хватает: `lambda`, `reduce` и списковое включение. Наверное, он занял бы последнее место на конкурсе популярности, потому что равно оскорбляет чувства тех, кто ненавидит `lambda`, и тех, кто презирает списковое включение, – а это чуть ли не вся публика.

Если вы собираетесь использовать `lambda`, то, пожалуй, нет причин прибегать к списковому включению, разве что для фильтрации, но здесь у нас не тот случай.

Вот мое решение, которое должно понравиться любителям `lambda`:

```
>>> functools.reduce(lambda a, b: a + b[1], my_list, 0)
60
```

Я не участвовал в этой ветке и не стал бы использовать этот код в реальной программе, потому что сам не большой поклонник `lambda`, но хотел показать, как можно решить эту задачу без спискового включения.

Первым ответил Фернандо Перес, создатель IPython, который привлек внимание к тому, что NumPy поддерживает *n*-мерные массивы и *n*-мерные срезы:

```
>>> import numpy as np
>>> my_array = np.array(my_list)
>>> np.sum(my_array[:, 1])
60
```

Мне кажется, что решение Переса очень изящное, но Гай Миддлтон выбрал другое, принадлежащее Полу Рубину и Скипу Монтанаро:

```
>>> import operator
>>> functools.reduce(operator.add, [sub[1] for sub in my_list], 0)
60
```

Затем Эван Симпсон спросил «А это чем плохо?»:

[1] Я немного изменил код для включения в книгу: в 2003 году функция `reduce` была встроенной, но в Python 3 ее нужно импортировать. Кроме того, я заменил имена `x` и `y` на `my_list` и `sub` (от sub-list).

```
>>> total = 0
>>> for sub in my_list:
... total += sub[1]
...
>>> total
60
```

Многие согласились, что это решение вполне в духе Python. Алекс Мартелли даже осмелился предположить, что так написал бы сам Гвидо.

Мне нравится код Эвана Симпсона, впрочем, как и комментарий к нему Дэвида Эпштейна:

> Если вы хотите просуммировать список элементов, то следует так и писать: «сумма списка элементов», а не «перебрать все элементы, завести еще одну переменную *t* и выполнить последовательность сложений». Зачем вообще нужны языки высокого уровня, если не для того, чтобы мы могли выразить свои намерения на высоком уровне – и пусть язык сам позаботится о том, какие низкоуровневые операции нужны для их реализации?

Затем вновь возник Алекс Мартелли с таким предложением:

> «Сумма» нужна так часто, что я не возражал бы, если бы в Python появилась такая встроенная функция. Но, на мой взгляд, «reduce(operator.add, ...» – не самый лучший способ выразить эту идею (вообще-то, имея большой опыт работы с APL и будучи поклонником функционального программирования, я должен был бы заценить этот код – но вот не нравится, и все тут).

Алекс далее предлагает функцию `sum()`, которую сам же и написал. Она стала встроенной в версии Python 2.3, вышедшей спустя всего три месяца после этой беседы. И вот так синтаксис, который предпочел Алекс, стал нормой:

```
>>> sum([sub[1] for sub in my_list])
60
```

В конце следующего года (ноябрь 2004) в версии Python 2.4 появились генераторные выражения, которые, на мой взгляд, дали самый «питонический» ответ на вопрос Гая Миддлтона:

```
>>> sum(sub[1] for sub in my_list)
60
```

Этот код не только понятнее версии с `reduce`, но и позволяет избежать проблем, когда последовательность пуста: `sum([])` равно `0` – вот так всё просто.

В той же беседе Алекс Мартелли высказал мысль, что встроенная функция `reduce` в Python 2 приносит больше хлопот, чем преимуществ, потому что поощряет применение идиом, которые трудно объяснить. Он был очень убедителен: в результате в Python 3 эта функция перекочевала в модуль `functools`.

И тем не менее у функции `functools.reduce` есть свое место под солнцем. Она позволила написать метод `Vector.__hash__` способом, который лично я считаю вполне в духе Python.

Глава 13

Интерфейсы, протоколы и ABC

> Программируйте в соответствии с интерфейсом, а не реализацией.
>
> – Гамма, Хелм, Джонсон, Влиссидес, «Первый принцип объектно-ориентированного программирования»[1]

Объектно-ориентированное программирование – это об интерфейсах. Хотите понять, что делает тип в Python, – узнайте, какие методы он предоставляет (его интерфейс), как описано в разделе «Типы определяются тем, какие операции они поддерживают» главы 8.

В разных языках программирования есть один или несколько способов определения и использования интерфейсов. Начиная с версии Python 3.8 у нас таких способов четыре. Все они изображены на *карте типизации* (рис. 13.1).

Утиная типизация

Подход к типизации, по умолчанию принятый в Python с момента его возникновения. Мы изучаем утиную типизацию начиная с главы 1.

Гусиная типизация

Этот подход поддерживается абстрактными базовыми глазами (ABC) и существует начиная с версии Python 2.6. В его основе лежит сравнение объектов с ABC, выполняемой на этапе выполнения. *Гусиная типизация* – основная тема этой главы.

Статическая типизация

Традиционный подход, принятый в статически типизированных языках, в частности C и Java; поддерживается, начиная с версии Python 3.5, с помощью модуля `typing` и обеспечивается внешними программами проверки типов по правилам, описанным в документе PEP 484 «Type Hints» (*https://peps.python.org/pep-0484/*). Эта тема в данной главе не рассматривается. Ей посвящена большая часть главы 8, а также глава 15.

Статическая утиная типизация

Этот подход стал популярным благодаря языку Go; поддерживается подклассами класса `typing.Protocol`, появившегося в версии 3.8, и также проверяется внешними программами. Впервые мы встретились с ним в разделе «Статические протоколы» главы 8.

---

[1] Design Patterns: Elements of Reusable Object-Oriented Software, «Introduction», стр. 18.

Карта типизации

Четыре основных подхода к типизации, изображенных на рис. 13.1, дополняют друг друга: у каждого есть плюсы и минусы. Ни один не стоит отбрасывать с порога.

Рис. 13.1. В верхней половине описаны подходы к проверке типов во время выполнения, для которых нужен только сам интерпретатор Python, а в нижней – требующие внешнего средства проверки типов, например MyPy или IDE типа PyCharm. В левых квадрантах находятся схемы типизации, основанные на структуре объекта, т. е. предоставляемых объектом методов, и не зависящие от имени класса или суперклассов. Схемы в правых квадрантах зависят от явных имен типов: имени класса объекта или его суперклассов

Каждый из четырех подходов опирается на интерфейсы, но статическую типизацию можно реализовать – плохо – с помощью одних только конкретных типов, не прибегая к абстракциям вроде протоколов и абстрактных базовых классов. Эта глава посвящена утиной типизации, гусиной типизации и статической утиной типизации – схемам, в основе которых лежат интерфейсы.

Глава разбита на четыре крупных раздела – по числу квадрантов на карте типизации:

- «Два вида протоколов» – сравниваются две формы структурной типизации с помощью протоколов, представленных в левой части карты типизации;
- «Программирование уток» – углубленное рассмотрение обычной утиной типизации в Python, в частности вопроса о том, как сделать ее безопаснее, не жертвуя главным достоинством: гибкостью;
- «Гусиная типизация» – объясняется, как ABC помогают выполнить более строгую проверку типов во время выполнения. Это самый длинный раз-

дел не потому, что он важнее прочих, а потому, что об утиной типизации, статической утиной типизации и статической типизации написано и в других частях книги;
- «Статические протоколы» – рассматривается использование, реализация и проектирование подклассов `typing.Protocol`, полезных для статической и динамической проверок типов.

Что нового в этой главе

Эта глава существенно переработана и стала примерно на 24 % длиннее соответствующей главы 11 в первом издании. Хотя некоторые разделы и многие абзацы повторяются, появилось и немало нового материала. Ниже перечислены основные отличия:

- введение к главе и карта типизации (рис. 13.1) написаны заново. Это ключ к большей части нового материала в этой главе – и во всех остальных, где речь идет о типизации в Python ⩾ 3.8;
- в разделе «Два вида протоколов» рассматриваются сходства и различия динамических и статических протоколов;
- раздел «Защитное программирование и принцип быстрого отказа» в основных чертах повторяет материал из первого издания, но был доработан, и теперь заголовок лучше отражает его важность;
- раздел «Статические протоколы» совсем новый. Он основан на первоначальных сведениях, изложенных в разделе «Статические протоколы» главы 8;
- изменены диаграммы классов из модуля `collections.abc` на рис. 13.2, 13.3 и 13.4; теперь они включают абстрактный базовый класс `Collection`, появившийся в Python 3.6.

В первом издании книги был раздел, где всячески поощрялось использование ABC `numbers` для гусиной типизации. В разделе «ABC из пакета numbers и числовые протоколы» я объясняю, почему вместо этого следует использовать числовые статические протоколы из модуля `typing`, если вы планируете применять средства статической проверки типов наряду с проверками во время выполнения в стиле гусиной типизации.

Два вида протоколов

В информатике слово *протокол* имеет разный смысл в зависимости от контекста. Сетевой протокол, например HTTP, описывает, какие команды клиент может отправлять серверу: `GET`, `PUT`, `HEAD` и т. д. В разделе «Протоколы и утиная типизация» главы 12 мы видели, что объектный протокол определяет методы, которые объект должен предоставлять, чтобы выполнить свое предназначение. В примере `FrenchDeck` в главе 1 продемонстрирован объектный протокол последовательности: методы, благодаря которым объект Python ведет себя как последовательность.

Для реализации полного протокола, возможно, требуется написать довольно много методов, но зачастую достаточно реализовать только часть. Рассмотрим класс `Vowels` в примере 13.1.

Пример 13.1. Частичная реализация протокола последовательности – метода `__getitem__`

```
>>> class Vowels:
...     def __getitem__(self, i):
...         return 'AEIOU'[i]
...
>>> v = Vowels()
>>> v[0]
'A'
>>> v[-1]
'U'
>>> for c in v: print(c)
...
A
E
I
O
U >>> 'E' in v
True
>>> 'Z' in v
False
```

Реализации метода `__getitem__` достаточно для получения элементов по индексу, а также поддержки итерирования и оператора `in`. Специальный метод `__getitem__` – ключ к протоколу последовательности. Приведем выдержку из «Справочного руководства по Python/C API», раздел «Протокол последовательности» (*https://docs.python.org/3/c-api/index.html*):

`int PySequence_Check(PyObject *o)`

> Возвращает `1`, если объект реализует протокол последовательности, иначе `0`. Отметим, что функция возвращает `1` для классов Python, реализующих метод `__getitem__()`, если только они не являются подклассами `dict` [...].

Мы ожидаем, что последовательность поддерживает также метод `len()` путем реализации специального метода `__len__`. В классе `Vowels` нет метода `__len__`, тем не менее в некоторых контекстах он ведет себя как последовательность. И возможно, для ваших целей этого достаточно. Потому-то я и говорю, что протокол – это «неформальный интерфейс». Именно так к протоколам относится Smalltalk, первая объектно-ориентированная среда программирования, в которой этот термин употреблялся.

Если не считать страниц документации по Python, относящихся к сетевому программированию, то чаще всего слово «протокол» встречается в контексте этих неформальных интерфейсов.

Но теперь, после реализации документа PEP 544 «Protocols: Structural subtyping (static duck typing)» (*https://peps.python.org/pep-0544/*) в Python 3.8, у слова «протокол» появилось другое значение – близкое, но все же отличающееся. В разделе «Статические протоколы» главы 8 мы видели, что PEP 544 позволяет создавать подклассы `typing.Protocol` с целью определить, какие методы должен реализовать (или унаследовать) класс, чтобы не раздражать программу статической проверки типов.

Когда контекст требует точности, я буду употреблять следующие термины:

Динамический протокол

Неформальные протоколы, которые были в Python всегда. Динамические протоколы неявные, определяются соглашением и описаны в документации. Самые важные динамические протоколы поддерживаются самим интерпретатором Python и документированы в главе «Модель данных» (*https://docs.python.org/3/reference/datamodel.html*) справочного руководства.

Статический протокол

Протокол, определенный в документе PEP 544 «Protocols: Structural subtyping (static duck typing)», начиная с версии Python 3.8. У статического протокола имеется явное определение: подкласс `typing.Protocol`.

Между этими двумя видами есть два основных различия:

- объект может реализовывать только часть динамического протокола и при этом быть полезным; но чтобы удовлетворить статическому протоколу, объект должен предоставить все методы, объявленные в классе протокола, даже если некоторые из них программе не нужны;
- статические протоколы можно проверить с помощью программ статической проверки типов, динамические – нельзя.

У обоих видов протоколов есть важная общая характеристика: класс не обязан объявлять, что поддерживает протокол с некоторым именем, например путем наследования.

Помимо статических протоколов, Python предлагает еще один способ программно определить явный интерфейс: абстрактный базовый класс (ABC).

Далее в этой главе мы будем рассматривать динамические и статические протоколы, а также ABC.

Программирование уток

Начнем обсуждение динамических протоколов с двух самых важных в Python: протоколов последовательности и итерируемого объекта. Интерпретатор из кожи вон лезет, стремясь обработать объекты, предоставляющие даже минимальную реализацию этих протоколов. Это демонстрируется в следующем разделе.

Python в поисках следов последовательностей

Философия модели данных Python заключается в том, чтобы всемерно взаимодействовать с важнейшими динамическими протоколами. А уж если речь идет о последовательностях, то Python прилагает все усилия, соглашаясь работать даже с самыми простыми реализациями.

На рис. 13.2 показано формальное определение интерфейса `Sequence` в виде ABC. Интерпретатор Python и встроенные последовательности `list`, `str` и др. вообще не полагаются на ABC. Я использую его, только чтобы описать, что ожидается от полноценной последовательности `Sequence`.

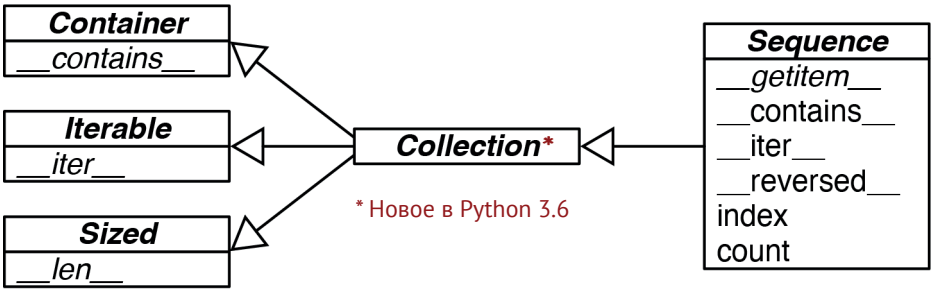


Рис. 13.2. UML-диаграмма абстрактного базового класса `Sequence` и связанных с ним классов из модуля `collections.abc`. Стрелки направлены от подклассов к суперклассам. Курсивом набраны имена абстрактных методов. До версии Python 3.6 не было ABC `Collection` – класс `Sequence` был прямым подклассом `Container`, `Iterable` и `Sized`

Большинство ABC в модуле `collections.abc` нужны для того, чтобы формализовать интерфейсы, реализованные встроенными объектами и неявно поддерживаемые интерпретатором, – то и другое существовало задолго до появления самих ABC. Но ABC полезны и как отправные точки для новых классов, и для поддержки явной проверки типов во время выполнения (как при *гусиной типизации*), и в качестве аннотаций типов для средств статической проверки типов.

Глядя на рис. 13.2, мы видим, что правильно написанный подкласс `Sequence` должен реализовывать методы `__getitem__` и `__len__` (унаследованный от `Sized`). Все остальные методы `Sequence` конкретные, поэтому подклассы могут унаследовать их реализации или предоставить собственные.

Теперь вспомним класс `Vowels` из примера 13.1. Он не наследует `abc.Sequence` и реализует только `__getitem__`.

Метода `__iter__` нет, но тем не менее экземпляры `Vowels` являются итерируемыми объектами, поскольку если Python находит метод `__getitem__` и не имеет ничего лучше, то он пытается обходить объект, вызывая этот метод с целочисленными индексами, начиная с `0`. Поскольку Python достаточно находчив, чтобы обойти экземпляры `Vowels`, ничто не мешает ему заставить работать оператор `in`, пусть даже метод `__contains__` отсутствует: нужно только произвести последовательный поиск и определить, имеется ли данный элемент.

Короче говоря, осознавая важность структур данных, обладающих свойствами последовательностей, Python ухитряется заставить итерирование и оператор `in` работать, вызывая метод `__getitem__` в случае, когда методы `__iter__` и `__contains__` отсутствуют.

Оригинальный класс `FrenchDeck` в главе 1 тоже не является подклассом `abc.Sequence`, но реализует оба метода протокола последовательности: `__getitem__` и `__len__`. См. пример 13.2.

Пример 13.2. Колода как последовательность карт (тот же код, что в примере 1.1)

```
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]
```

Ряд примеров из главы 1 работают, потому что Python специальным образом обрабатывает все, что отдаленно напоминает последовательность. Протокол итерируемого объекта в Python представляет собой крайнюю форму утиной типизации: интерпретатор пробует два разных метода обхода объектов.

Если быть точным, то описанные в этом разделе поведения реализованы в самом интерпретаторе, в основном на C. Они не зависят от методов ABC `Sequence`. Например, конкретные методы `__iter__` и `__contains__` в классе `Sequence` эмулируют встроенные поведения интерпретатора Python. Особо любознательные могут посмотреть исходный код этих методов в файле *Lib/_collections_abc.py* (*https://github.com/python/cpython/blob/31ceccb2c77854893f3a754aca04bedd74bedb10/Lib/_collections_abc.py#L870*).

А теперь рассмотрим еще один пример, подчеркивающий динамическую природу протоколов, а заодно показывающий, почему средства статической проверки типов не имеют ни единого шанса справиться с ними.

Партизанское латание как средство реализации протокола во время выполнения

Под партизанским латанием понимается динамическое изменение модуля, класса или функции во время выполнения – чтобы добавить новые возможности или исправить ошибки. Например, сетевая библиотека gevent по-партизански латает части стандартной библиотеки Python, чтобы реализовать облегченную конкурентность без потоков или `async/await`[1].

У класса `FrenchDeck` из примера 13.2 есть существенный изъян: колоду нельзя перетасовать. Много лет назад, впервые написав этот пример, я реализовал метод `shuffle`. Позже меня посетило питоническое озарение: если `FrenchDeck` ведет себя как последовательность, то ему не нужен собственный метод `shuffle`, потому что уже имеется функция `random.shuffle`, в документации по которой (*https://docs.python.org/ 3/library/random.html#random.shuffle*) написано: «Перетасовывает последовательность *x* на месте».

Стандартная функция `random.shuffle` используется следующим образом:

[1] В статье Википедии «Monkey patch» (*https://en.wikipedia.org/wiki/Monkey_patch*) есть забавный пример на Python.

```
>>> from random import shuffle
>>> l = list(range(10))
>>> shuffle(l)
>>> l
[5, 2, 9, 7, 8, 3, 1, 4, 0, 6]
```

Если следовать устоявшимся протоколам, то будет больше шансов воспользоваться кодом, уже имеющимся в стандартной библиотеке или написанным кем-то еще, – благодаря утиной типизации.

Но, попытавшись перетасовать объект `FrenchDeck`, мы получим исключение (пример 11.5).

Пример 13.3. `random.shuffle` не может работать с объектом `FrenchDeck`

```
>>> from random import shuffle
>>> from frenchdeck import FrenchDeck
>>> deck = FrenchDeck()
>>> shuffle(deck)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../random.py", line 265, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'FrenchDeck' object does not support item assignment
```

В сообщении ясно говорится: «Объект `'FrenchDeck'` не поддерживает присваивание элементу». Проблема в том, что `shuffle` должна иметь возможность переставить два элемента коллекции, а класс `FrenchDeck` реализует только протокол *неизменяемой* последовательности. Изменяемая последовательность должна также предоставлять метод `__setitem__`.

Поскольку Python – динамический язык, мы можем устранить проблему прямо во время выполнения, даже в интерактивной оболочке. В примере 11.6 показано, как это сделать.

Пример 13.4. Партизанское латание класса `FrenchDeck` с целью сделать его изменяемым и совместимым с функцией `random.shuffle` (продолжение примера 13.3)

```
>>> def set_card(deck, position, card): ❶
... deck._cards[position] = card
...
>>> FrenchDeck.__setitem__ = set_card ❷
>>> shuffle(deck) ❸
>>> deck[:5]
[Card(rank='3', suit='hearts'), Card(rank='4', suit='diamonds'), Card(rank='4',
suit='clubs'), Card(rank='7', suit='hearts'), Card(rank='9', suit='spades')]
```

❶ Создать функцию, которая принимает аргументы `deck`, `position` и `card`.
❷ Присвоить эту функцию атрибуту `__setitem__` класса `FrenchDeck`.
❸ Теперь объект `deck` можно перетасовать, потому что класс `FrenchDeck` поддерживает обязательный метод протокола изменяемой последовательности.

Сигнатура метода `__setitem__` определена в разделе 3.3.6 «Эмуляция контейнерных типов» справочного руководства по языку Python (*https://docs.python.org/3/reference/ datamodel.html#emulating-container-types*). В данном случае мы на-

звали аргументы `deck`, `position`, `card` – а не `self`, `key`, `value`, как в руководстве, – чтобы показать, что любой метод Python изначально является простой функцией, а имя `self` для первого аргумента – не более чем соглашение. В сеансе оболочки это нормально, но в исходном файле Python гораздо лучше использовать предлагаемые в документации имена `self`, `key` и `value`.

Трюк состоит в том, что функция `set_card` знает о наличии в объекте `deck` атрибута с именем `_cards`, который должен быть изменяемой последовательностью. После этого мы присоединяем функцию `set_card` к классу `FrenchDeck` в качестве специального метода `__setitem__`. Это пример *партизанского латания* (monkey patching): изменения класса или модуля во время выполнения без модификации исходного кода. Техника весьма действенная, но код, в котором она используется, оказывается очень тесно связан с латаемой программой и зачастую даже вмешивается в ее закрытые и недокументированные части.

Помимо партизанского латания, в примере 11.6 иллюстрируется динамичность протоколов: функции `random.shuffle` безразлично, какие аргументы ей переданы, лишь бы объект реализовал часть протокола изменяемой последовательности. Не важно даже, получил ли объект необходимые методы «при рождении» или каким-то образом приобрел их позже.

Не следует думать, что утиная типизация абсолютно небезопасна или что ее ужасно трудно отлаживать. В следующем разделе показано несколько полезных способов обнаружить динамический протокол, не прибегая к явным проверкам.

Защитное программирование и принцип быстрого отказа

Защитное программирование – как контраварийное вождение: набор практических навыков, повышающих безопасность при столкновении с беспечными программистами – или водителями.

Многие ошибки можно отловить только во время выполнения, даже в популярных статически типизированных языках[1]. В динамически типизированном языке принцип быстрого отказа – прекрасный совет по созданию более безопасных и удобных для сопровождения программ. Быстрый отказ означает, что нужно возбудить исключение как можно раньше, например отвергать недопустимые аргументы в самом начале тела функции.

Приведу пример: разрабатывая код, который принимает последовательность элементов, обрабатываемую как `list`, не проверяйте, что аргумент имеет тип `list`. Вместо этого попробуйте сразу же построить `list` из аргумента. Пример такого подхода дает метод `__init__` в примере 13.10 ниже.

```
    def __init__(self, iterable):
        self._balls = list(iterable)
```

Это позволит сделать код более гибким, потому что конструктор `list()` готов обрабатывать любой итерируемый объект, помещающийся в память. Если аргумент не является итерируемым, что отказ будет быстрым, а сообщение об ошибке в исключении `TypeError` вполне недвусмысленным и прямо в той точке, где объект инициализируется. Если нужно что-то более явное, то можете обернуть `list()` операторными скобками `try/except`, чтобы настроить сообщение

[1] Именно поэтому необходимо автоматизированное тестирование.

об ошибке, но я бы использовал такой дополнительный код только при работе с внешними API, потому что для лиц, отвечающих за сопровождение кодовой базы, ошибка будет очевидной. В любом случае проблемный вызов будет находиться в конце обратной трассы вызовов, так что ошибку будет легко исправить. Если вы не станете обнаруживать недопустимый аргумент в конструкторе класса, то программа рухнет позже, когда какому-то методу потребуется произвести операцию с `self._balls` и окажется, что это не `list`. В таком случае найти исходную причину будет труднее.

Разумеется, передавать аргумент конструктору `list()` не стоит, если данные не должны копироваться, – то ли потому, что они слишком велики, то ли потому, что функция, в силу замысла, должна изменять его на месте, поскольку того требует вызывающая сторона, как в случае `random.shuffle`. В подобном случае рекомендуется выполнить проверку во время выполнения, например `isinstance(x, abc.MutableSequence)`.

Если вы боитесь получить бесконечный генератор – не слишком частая проблема, – то для начала вызовите метод `len()` аргумента. Тем самым вы отсечете итераторы, но сможете безопасно обработать кортежи, массив и другие существующие и будущие классы, которые полностью реализуют интерфейс `Sequence`. Вызов `len()` обычно обходится очень дешево, а если аргумент недопустим, то ошибка произойдет немедленно.

С другой стороны, если вы готовы принять любой итерируемый объект, то как можно быстрее вызовите метод `iter(x)`, чтобы получить итератор, как будет показано в разделе «Почему последовательности являются итерируемыми объектами: функция iter». И снова, если объект `x` неитерируемый, то отказ произойдет быстро и отлаживать исключение будет легко.

В описанных выше случаях аннотация типа могла бы обнаружить раньше некоторые, но не все ошибки. Напомним, что тип `Any` *совместим с* любым другим типом. Механизм вывода типа может пометить переменную типом `Any`. В таком случае средство проверки типов бессильно. Кроме того, аннотации типов не проверяются во время выполнения. Быстрый отказ – последняя линия обороны.

Защитный код, в котором используется утиная типизация, может также включать логику обработки разных типов без привлечения проверок `isinstance()` или `hasattr()`.

Например, можно было бы эмулировать обработку аргумента `field_names` в `collections.namedtuple`. Напомним, что в качестве `field_names` может фигурировать одна строка идентификаторов, разделенных пробелами или запятыми, или последовательность идентификаторов. В примере 13.5 показано, как я сделал бы это с помощью утиной типизации.

Пример 13.5. Применение утиной типизации для обработки строки или итерируемого объекта строк

```
try: ❶
    field_names = field_names.replace(',', ' ').split() ❷
except AttributeError: ❸
    pass ❹
field_names = tuple(field_names)
if not all(s.isidentifier() for s in field_names): ❻
    raise ValueError('field_names must all be valid identifiers')
```

❶ Предполагаем, что это строка (проще попросить прощения, чем испрашивать разрешение).
❷ Заменяем запятые пробелами и разбиваем образовавшуюся строку, получая список имен.
❸ Увы, `field_names` не крякает, как `str`..., – то ли метода `.replace` нет, то ли он возвращает нечто такое, к чему нельзя применить `.split`.
❹ Если было возбуждено исключение `AttributeError`, то аргумент `field_names` не является строкой `str`, поэтому предполагаем, что он уже является итерируемым объектом, содержащим имена.
❺ Чтобы убедиться в его итерируемости и заодно получить внутреннюю копию, создаем кортеж из того, что имеем. Кортеж `tuple` компактнее, чем `list`, а заодно предотвращает изменение имен по ошибке.
❽ Используем метод `str.isidentifier`, гарантирующий уникальность имен.

Пример 13.5 демонстрирует одну ситуацию, когда утиная типизация оказывается более выразительной, чем статические аннотации типов. Невозможно записать аннотацию, которая говорила бы, что «`field_names` должен быть строкой идентификаторов, разделенных пробелами или запятыми». Это релевантная часть сигнатуры `namedtuple` в typeshed (полный код см. в файле *stdlib/3/collections/__init__.pyi* по адресу *https://github.com/python/typeshed/blob/ 24afb531ffd07083d6a74be917342195062f7277/stdlib/collections/__init__.pyi*):

```
def namedtuple(
    typename: str,
    field_names: Union[str, Iterable[str]],
    *,
    # остальная часть сигнатуры опущена
```

Как видите, аргумент `field_names` аннотирован как `Union[str, Iterable[str]]`, что неплохо, но недостаточно для отлавливания всех возможных ошибок.

Сделав обзор динамических протоколов, обратимся к более явной форме проверки типов во время выполнения: гусиной типизации.

Гусиная типизация

> Абстрактный класс предоставляет интерфейс.
>
> – Бьярн Страуструп, создатель C++[1]

В языке Python нет ключевого слова `interface`. Мы используем абстрактные базовые классы (ABC), чтобы определить интерфейсы для явной проверки типов во время выполнения. Они также поддерживаются программами статической проверки типов.

В «Глоссарии Python» статья, посвященная абстрактному базовому классу (*https://docs.python.org/3/glossary.html#term-abstract-base-class*), объясняет, какую ценность они привносят в языки с утиной оптимизацией:

> Абстрактные базовые классы дополняют утиную типизацию, поскольку предоставляют способ определять интерфейс в случаях, когда другие приемы, например `hasattr()`, выглядели бы неуклюже или содержали бы тонкие

[1] *Бьярн Страуструп.* Дизайн и эволюция C++. ДМК, 2000. С. 284.

ошибки (например, с помощью магических методов). ABC привносят виртуальные подклассы, т. е. классы, которые не наследуют другому классу, но тем не менее распознаются функциями `isinstance()` и `issubclass()`; см. описание модуля `abc` в документации[1].

Гусиная типизация – это подход к проверке типов во время выполнения, основанный на применении ABC. Предоставляю слово Алексу Мартелли.

Я очень благодарен своим друзьям Алексу Мартелли и Анне Равенскрофт. Я показал им план этой книги на конференции OSCON 2013, и они посоветовали мне предложить ее издательству O'Reilly для публикации. Впоследствии оба написали дотошные технические рецензии. Алекс и так уже был самым цитируемым персонажем в этой книге, а затем еще и предложил нижеследующее эссе. Вот оно, Алекс!

Водоплавающие птицы и ABC

Алекс Мартелли

В Википедии (*http://en.wikipedia.org/wiki/Duck_typing#History*) мне приписывают честь распространения полезного мема и эффектного выражения «утиная типизация» (т. е. игнорирование фактического типа объекта и акцент на то, чтобы объект реализовывал методы с именами, сигнатурами и семантикой, требуемыми для конкретного применения).

В Python это сводится в основном к тому, чтобы избегать использования функции `isinstance` для проверки типа объекта (я уже не говорю о еще более вредном подходе: проверке вида `type(foo) is bar`, которую следует предать анафеме, потому что она препятствует даже простейшим формам наследования!).

В целом утиная типизация остается весьма полезной во многих контекстах, однако есть и много других, где со временем выработался иной, более предпочтительный подход. Отсюда и начинается наш рассказ…

Уже для многих поколений классификация по родам и видам (в том числе и семейства водоплавающих, известного под названием Anatidae) основывается главным образом на фенетике – когда во главу угла ставится сходство морфологии и поведения… в общем, на *наблюдаемых* характеристиках. Аналогия с «утиной типизацией» была очень сильной.

Однако в ходе параллельной эволюции зачастую сходные характеристики, как морфологические, так и поведенческие, оказываются у видов, которые фактически не связаны друг с другом, но просто эволюционировали в похожих, хотя и разных, экологических нишах. Подобное «случайное сходство» встречается и в программировании. Для иллюстрации возьмем классический пример из ООП:

```
class Artist:           # художник
    def draw(self): ... # рисовать
class Gunslinger:       # стрелок
    def draw(self): ... # выхватить револьвер
```

[1] По состоянию на 18 октября 2020 года.

```
class Lottery:          # лотерея
    def draw(self): ... # тянуть билетик
```

Очевидно, одного лишь существования метода `draw` без аргументов далеко недостаточно, чтобы убедить нас в том, что два объекта `x` и `y` такие, что допустимы вызовы `x.draw()` и `y.draw()`, являются хоть в какой-то степени взаимозаменяемыми или абстрактно эквивалентными, – из допустимости подобных вызовов нельзя сделать никаких выводов о схожести семантики. Понадобится опытный программист, который взялся бы уверенно *подтвердить*, что такая эквивалентность имеет место на каком-то уровне!

В биологии (и других дисциплинах) эта проблема стала причиной появления (а во многих отношениях и преобладания) подхода, альтернативного фенетике, а именно *кладистики* – классификации с упором на характеристики, унаследованные от общих предков, а не появившиеся в результате независимой эволюции. (Дешевая и быстрая методика секвенирования ДНК может сделать кладистику практически полезной в гораздо большем числе случаев, чем сейчас.)

Например, гуси-пеганки и утки-пеганки (которые раньше в классификации стояли ближе к другим гусям и уткам) теперь помещены в подсемейство Tadornidae (откуда следует, что они ближе друг к другу, чем к другим представителям семейства Anatidae, поскольку имеют общего предка). Кроме того, анализ ДНК показал, что белокрылая каролинская утка не так близка к мускусной утке (которая является уткой-пеганкой), как можно было бы предположить по внешнему виду и поведению. Поэтому классификация каролинский утки была изменена, ее вообще исключили из подсемейства и выделили в отдельный род!

Важно ли это? Все зависит от контекста! Если нужно решить, как лучше приготовить водоплавающую птицу, которую вы уже добыли, то наблюдаемые характеристики (не все – скажем, наличие плюмажа в этом случае роли не играет) и прежде всего структура мяса и вкусовые качества (старомодная фенетика!) гораздо важнее кладистики. Но в других вопросах, например в отношении восприимчивости к различным патогенным организмам (следует ли пытаться выращивать птиц в неволе или сохранять их в дикой природе), близость ДНК может оказаться гораздо важнее…

Итак, в силу наличия отдаленной аналогии с таксономической революцией в мире водоплавающих птиц я рекомендую дополнить старую добрую *утиную типизацию* (не вовсе заменить – в некоторых контекстах она нам еще послужит) … *гусиной типизацией*!

Гусиная типизация означает следующее: вызов `isinstance(obj, cls)` теперь считается приемлемым… при условии, что `cls` – абстрактный базовый класс, т. е. метаклассом `cls` является `abc.ABCMeta`.

В модуле `collections.abc` можно найти немало полезных абстрактных классов (они есть также в модуле `numbers` из стандартной библиотеки Python)[1].

[1] Разумеется, вы можете определить и свои ABC, но я не советую это делать никому, кроме самых опытных питонистов, равно как не советую определять свои метаклассы… и даже для этих «самых опытных питонистов», знающих обо всех потаенных уголках и темных закоулках языка, это инструменты не для каждодневного использования. Эти средства «углубленного метапрограммирования» предназначены авторам каркасов широкого назначения, которые предположительно будут независимо развивать многочисленные не связанные между собой команды разработчиков. В общем, они могут понадобиться менее чем 1 % «самых опытных питонистов»! – *А. М.*

Из многих концептуальных преимуществ ABC по сравнению с конкретными классами (например, Скотт Мейер в своей книге «Более эффективный C++», совет 33 – *http://ptgmedia.pearsoncmg.com/images/020163371x/items/item33.html*, – говорит, что «все нелистовые классы должны быть абстрактными») выделим одно практически важное достоинство ABC в Python: метод класса `register`, который дает возможность конечному пользователю «объявить» некоторый класс «виртуальным» подклассом ABC (для этого зарегистрированный класс должен удовлетворять требованиям ABC к имени и сигнатуре и, что еще важнее, подразумеваемому семантическому контракту, но его необязательно разрабатывать с учетом ABC и, в частности, не требуется наследовать ему!). Это большой шаг на пути к устранению жесткости и сильной сцепленности, из-за которых к наследованию следует относиться с куда большей настороженностью, чем позволяет себе большинство программирующих на ОО-языках...

Иногда даже и регистрировать класс не нужно, чтобы ABC распознал его как подкласс!

Так бывает в случае ABC, существующих только ради нескольких специальных методов. Например:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
```

Как видим, `abc.Sized` распознал `Struggle` как свой «подкласс» безо всякой регистрации просто потому, что для этого необходимо только наличие специального метода `__len__` (предполагается, что он реализован правильно с точки зрения синтаксиса – вызывается без аргументов – и семантики – возвращает неотрицательное целое число, интерпретируемое как «длина» объекта; программа, которая реализует специальный метод, например `__len__`, с какими-то другими, несогласованными синтаксисом и семантикой, в любом случае обречена на куда более серьезные проблемы).

Итак, вот мое напутствие: реализуя класс, который воплощает концепции, представленные в ABC из модуля `numbers`, `collections.abc` или какого-то другого каркаса, либо делайте его подклассом ABC (если необходимо), либо регистрируйте. В начале программы, использующей библиотеку или каркас, где определяются классы, для которых это не сделано, выполняйте регистрацию самостоятельно. Затем, если потребуется проверить, что аргумент (чаще всего это необходимо как раз для аргументов) является, к примеру, «последовательностью», пишите:

```
isinstance(the_arg, collections.abc.Sequence)
```

И *не* определяйте свои ABC (или метаклассы) в производственном коде. Если вам кажется, что без этого не обойтись, держу пари, что это, скорее всего, желание поскорее забить гвоздь, раз уж в руках молоток, – вам (и тем, кому предстоит сопровождать вашу программу) будет куда комфортнее иметь дело с прямолинейным и простым кодом, где нет таких глубин. Valē!

Таким образом, *гусиная типизация* подразумевает:

- порождение подклассов ABC, чтобы было ясно, что мы реализуем ранее определенный интерфейс;
- проверку во время выполнения с указанием ABC вместо конкретных классов в качестве второго аргумента функций `isinstance` и `issubclass`.

Алекс подчеркивает, что наследование ABC не сводится к реализации необходимых методов, это еще и четкое заявление о намерениях разработчика. Такое намерение можно сделать явным также путем регистрации виртуального подкласса.

Подробнее использование `register` рассматривается в разделе «Виртуальный класс ABC» ниже в этой главе. А пока приведу краткий пример. Пусть имеется класс `FrenchDeck`, и я хочу проверять его тип следующим образом: `issubclass(FrenchDeck, Sequence)`. Для этого я могу сделать его виртуальным подклассом ABC `Sequence`:

```
from collections.abc import Sequence
Sequence.register(FrenchDeck)
```

Использование функций `isinstance` и `issubclass` выглядит уже не столь одиозным, если сравнивать тип с ABC, а не с конкретными классами. При использовании в сочетании с конкретными классами проверка типа ограничивает полиморфизм – существенную часть объектно-ориентированного программирования. Но с появлением ABC проверки становятся более гибкими. Ведь даже если компонент не является подклассом ABC, но реализует требуемые методы, его всегда можно зарегистрировать постфактум, так что он пройдет эти явные проверки типа.

Однако и при использовании ABC нужно помнить, что злоупотребление функцией `isinstance` может быть признаком «дурно пахнущего кода» – плохо спроектированной объектно-ориентированной программы.

Обычно *не* должно быть цепочек предложений `if/elif/elif`, в которых с помощью `insinstance` определяется тип объекта и в зависимости от него выполняются те или иные действия; для этой цели следует использовать полиморфизм, т. е. проектировать классы, так чтобы интерпретатор сам вызывал правильные методы, а не «зашивать» логику диспетчеризации в блоки `if/elif/elif`.

С другой стороны, нет возражений против использования `insinstance` для сравнения с типом ABC, если требуется убедиться в соблюдении контракта: «Эй, чтобы меня вызывать, ты должен реализовать то-то и то-то», как выразился технический рецензент Леннарт Регебро. Особенно это полезно в системах, основанных на архитектуре плагинов. За пределами каркасов утиная типизация обычно проще и дает большую гибкость, чем проверка типов.

Наконец, в своем эссе Алекс неоднократно подчеркивает, что не стоит усердствовать в создании ABC. Злоупотребление ABC вынудило бы выполнять ненужные церемонии в языке, завоевавшем популярность своей практичностью и прагматичностью. В своей рецензии на эту книгу Алекс написал:

> ABC предназначены для инкапсуляции очень общих концепций, абстракций, характерных для каркаса, – таких вещей, как «последовательность» или «точное число». [Читателям], скорее всего, не придется писать новые ABC, а лишь правильно использовать существующие. В 99.9 % случаев этого будет достаточно для получения всех преимуществ без риска спроектировать что-то не то.

Ну а теперь посмотрим, как гусиная типизация выглядит на практике.

Создание подкласса ABC

Следуя совету Мартелли, мы воспользуемся существующим ABC `collections.MutableSequence`, перед тем как изобретать свой собственный. В примере 13.6 класс `FrenchDeck2` явно объявлен подклассом `collections.MutableSequence`.

Пример 13.6. frenchdeck2.py: `FrenchDeck2`, подкласс `collections.MutableSequence`

```
from collections import namedtuple, abc

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

class FrenchDeck2(abc.MutableSequence):
    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]

    def __setitem__(self, position, value): ❶
        self._cards[position] = value

    def __delitem__(self, position): ❷
        del self._cards[position]

    def insert(self, position, value): ❸
        self._cards.insert(position, value)
```

❶ Метод `__setitem__` – все, что нам нужно для поддержки тасования…
❷ … но чтобы создать подкласс `MutableSequence`, нам придется реализовать также `__delitem__` – абстрактный метод, определенный в этом ABC.
❸ Еще необходимо реализовать `insert`, третий абстрактный метод `MutableSequence`.

На этапе импорта (когда модуль *frenchdeck2.py* загружается и компилируется) Python не проверяет, реализованы ли абстрактные методы. Это происходит только на этапе выполнения, когда мы пытаемся создать объект `FrenchDeck2`. И тогда, если абстрактный метод не реализован, мы получим исключение `TypeError` с сообщением вида `«Can't instantiate abstract class FrenchDeck2 with abstract methods __delitem__, insert»`. Вот поэтому мы и обязаны реализовать методы `__delitem__` и `insert`, хотя в наших примерах класс `FrenchDeck2` в них и не нуждается; ничего не поделаешь – абстрактный базовый класс `MutableSequence` требует.

Как показано на рис. 13.3, не все методы ABC `Sequence` и `MutableSequence` абстрактны.

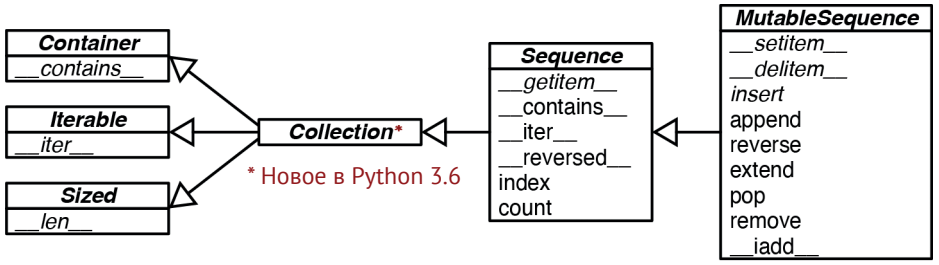


Рис. 13.3. UML-диаграмма класса `MutableSequence` и его суперклассов из модуля `collections.abc`. Стрелки направлены от подклассов к суперклассам. Курсивом набраны имена абстрактных классов и методов

Чтобы сделать `FrenchDeck2` подклассом `MutableSequence`, пришлось реализовать методы `__delitem__` и `insert`, которые в моих примерах не нужны. В награду за труды `FrenchDeck2` наследует пять конкретных методов от `Sequence`: `__contains__`, `__iter__`, `__reversed__`, `index` и `count`. От `MutableSequence` он получает еще шесть методов: `append`, `reverse`, `extend`, `pop`, `remove` и `__iadd__`; последний поддерживает оператор `+=` для конкатенации на месте.

Конкретные методы в каждом ABC из модуля `collections.abc` реализованы в терминах открытого интерфейса класса, поэтому для работы им не нужны никакие знания о внутренней структуре экземпляров.

Кодировщику конкретного подкласса иногда приходится переопределять методы, унаследованные от ABC, предоставляя более эффективную реализацию. Например, унаследованный метод `__contains__` просматривает всю последовательность, но если в конкретной последовательности элементы всегда отсортированы, то можно написать более быстрый вариант `__contains__`, который будет производить двоичный поиск с помощью функции `bisect`. См. раздел «Средства работы с упорядоченными последовательностями в модуле bisect» (*https://www.fluentpython.com/extra/ordered-sequences-with-bisect/*) на сайте *fluentpython.com*.

Чтобы работать с ABC, нужно знать, что есть в нашем распоряжении. Далее мы рассмотрим ABC коллекций.

ABC в стандартной библиотеке

Начиная с версии Python 2.6 ABC включены в стандартную библиотеку. Большая их часть определена в модуле `collections.abc`, но есть и другие. Например, ABC можно найти в пакетах `numbers` и `io`. Но самые употребительные находятся в `collections.abc`.

В стандартной библиотеке есть два модуля с именем `abc`. Мы сейчас говорим о модуле `collections.abc`. Чтобы уменьшить время загрузки, в версии Python 3.4 он находится не в пакете `collections`, а в файле *Lib/_collections_abc.py* (*https://github.com/python/cpython/blob/main/Lib/_collections_abc.py*), поэтому импортируется отдельно от `collections`. Другой модуль `abc` называется

просто `abc` (т. е. *Lib/abc.py* – *https://hg.python.org/cpython/file/3.4/Lib/abc.py*), в нем определен класс `abc.ABC`. Все ABC зависят от этого класса, но импортировать его самостоятельно нужно только при создании нового ABC.

На рис. 13.4 приведена сокращенная UML-диаграмма классов (без имен атрибутов), на которой показаны все 17 ABC, определенных в модуле `collections.abc`. В официальной документации по модулю `collections.abc` имеется симпатичная таблица (*https://docs.python.org/3/library/collections.abc.html#collections-abstract-base-classes*), в которой перечислены ABC, их взаимосвязи, а также абстрактные и конкретные методы (так называемые «методы-примеси»). На рис. 13.4 мы видим немало примеров множественного наследования. Множественному наследованию посвящена большая часть главы 14, а пока скажем лишь, что в случае абстрактных базовых классов оно обычно не составляет проблемы[1].

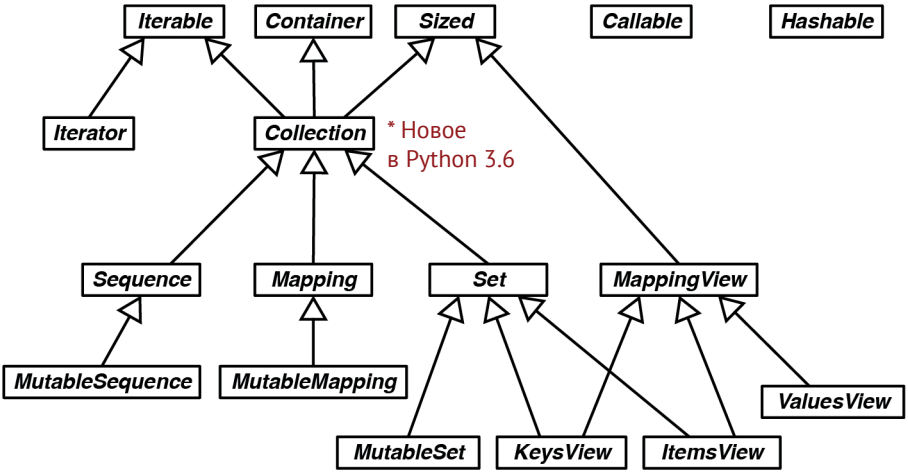


Рис. 13.4. UML-диаграмма абстрактных базовых классов из модуля `collections.abc`

Коротко рассмотрим группы классов на рис. 13.4.

`Iterable`, `Container`, `Sized`

Любая коллекция должна либо наследовать этим ABC, либо реализовывать совместимые протоколы. Класс `Iterable` поддерживает итерирование методом `__iter__`, `Container` поддерживает оператор `in` методом `__contains__`, а `Sized` – функцию `len()` методом `__len__`.

`Collection`

У этого ABC нет собственных методов, но он был добавлен в версию Python 3.6, чтобы было проще создавать подклассы, наследующие `Iterable`, `Container` и `Sized`.

[1] Множественное наследование было *сочтено вредным* и исключено из языка Java. Исключение было сделано для интерфейсов: в Java интерфейс может расширять несколько интерфейсов, а класс – реализовывать несколько интерфейсов.

`Sequence`, `Mapping`, `Set`

Это основные типы неизменяемых коллекций, и у каждого есть изменяемый подкласс. Детальная диаграмма класса `MutableSequence` показана на рис. 13.2, а диаграммы классов `MutableMapping` и `MutableSet` приведены в главе 3 (рис. 3.1 и 3.2).

`MappingView`

В Python 3 объекты, возвращенные методами отображения `.items()`, `.keys()` и `.values()`, наследуют классам `ItemsView`, `KeysView` и `ValuesView` соответственно. Первые два также наследуют богатый интерфейс класса `Set` со всеми операторами, которые были описаны в разделе «Операции над множествами» главы 3.

`Iterator`

Отметим, что класс `Iterator` является подклассом `Iterable`. Мы еще вернемся к этому вопросу в главе 17.

`Callable`, `Hashable`

Это не коллекции, но `collections.abc` был первым пакетом в стандартной библиотеке, где были определены ABC, а эти два класса казались достаточно важными, чтобы включить их. Их основное назначение – поддержка проверки типов объектов, которые должны быть вызываемыми или хешируемыми.

Чтобы определить, является ли объект вызываемым, удобнее воспользоваться встроенной функцией `callable(obj)`, чем писать `insinstance(obj, Callable)`.

Если `insinstance(obj, Hashable)` возвращает `False`, то можно точно сказать, что `obj` не хешируемый. Но если она возвращает `True`, то возможен ложноположительный результат. В следующей врезке объясняется, почему.

Функция isinstance для Hashable и Iterable может вводить в заблуждение

Результаты применения `isinstance` и `issubclass` к ABC `Hashable` и `Iterable` легко интерпретировать неправильно. Если `isinstance(obj, Hashable)` возвращает `True`, то это лишь означает, что класс `obj` реализует или наследует метод `__hash__`. Но если `obj` является кортежем `tuple`, содержащим нехешируемые элементы, то `obj` не является хешируемым, несмотря на положительный результат проверки с помощью `isinstance`. Технический рецензент Юрген Гмах указал, что утиная типизация дает самый надежный способ определить, является ли экземпляр хешируемым: вызвать `hash(obj)`. Этот вызов приведет к исключению `TypeError`, если `obj` не хешируемый.

С другой стороны, даже если `isinstance(obj, Iterable)` возвращает `False`, у Python все равно остается возможность обойти `obj` с помощью метода `__getitem__`, который применяется для получения элементов с индексами, начиная с 0. Мы видели, как это делается в главе 1 и в разделе «Python в поисках следов последовательностей». В документации по классу `collections.abc.Iterable` говорится:

> Единственно надежный способ определить, является ли объект итерируемым, – вызвать `iter(obj)`.

Познакомившись с некоторыми имеющимися ABC, попрактикуемся в гусиной типизации, для чего реализуем ABC с нуля и воспользуемся им. Цель не в том, чтобы очертя голову бросаться писать ABC, а в том, чтобы научиться читать исходный код ABC, находящихся в стандартной библиотеке и в других пакетах.

Определение и использование ABC

Следующее предупреждение присутствовало в главе «Интерфейсы» первого издания книги:

> ABC, подобно дескрипторам и метаклассам, предназначены для разработки каркасов. Поэтому лишь малая часть пишущих на Python может создавать ABC, не налагая ненужных ограничений на своих коллег-программистов и не заставляя их делать бессмысленную работу.

Теперь у ABC появились дополнительные возможности применения в аннотациях типов для поддержки статической типизации. В разделе «Абстрактные базовые классы» главы 8 мы уже говорили, что использование ABC вместо конкретных типов для аннотирования типов аргументов функции дает больше гибкости вызывающей стороне.

Чтобы оправдать создание абстрактного базового класса, нам необходим контекст для использования его в качестве точки расширения в каком-то каркасе. Возьмем такой контекст: пусть требуется отображать на сайте или в мобильном приложении рекламные объявления в случайном порядке, но при этом не повторять никакое объявление, пока не будут показаны все остальные из имеющегося набора. Допустим, мы разрабатываем систему управления рекламой под названием `ADAM`. Одно из требований – поддержать предоставляемые пользователем классы случайного выбора без повторений[1]. Чтобы у пользователей `ADAM` не было сомнений, что понимается под «случайным выбором без повторений», мы определим ABC.

В литературе по структурам данных «стек» и «очередь» описывают абстрактные интерфейсы в терминах физической организации объектов. Я последую этой традиции и назову наш ABC, руководствуясь следующей метафорой из реального мира: барабаны для бинго и лотереи – это машины, предназначенные для случайного выбора элемента из конечного множества, без повторений, до полного исчерпания множества.

Наш ABC будет называться `Tombola`, это итальянское название игры в бинго и опрокидывающегося контейнера, в котором перемешиваются номера.

В ABC `Tombola` определены четыре метода. Два из них абстрактны:

`.load(…)`

Поместить элементы в контейнер.

`.pick()`

Извлечь случайный элемент из контейнера и вернуть его.

И есть еще два конкретных метода:

[1] Быть может, клиент захочет подвергнуть рандомизатор аудиту или рекламное агентство решит предоставить какой-то особо хитрый рандомизатор. Заранее никогда не скажешь…

`.loaded()`

Вернуть `True`, если в контейнере имеется хотя бы один элемент.

`.inspect()`

Вернуть кортеж `tuple`, составленный из элементов, находящихся в контейнере, не изменяя его содержимого (внутреннее упорядочение не сохраняется).

На рис. 13.5 показан ABC `Tombola` и три его конкретные реализации.

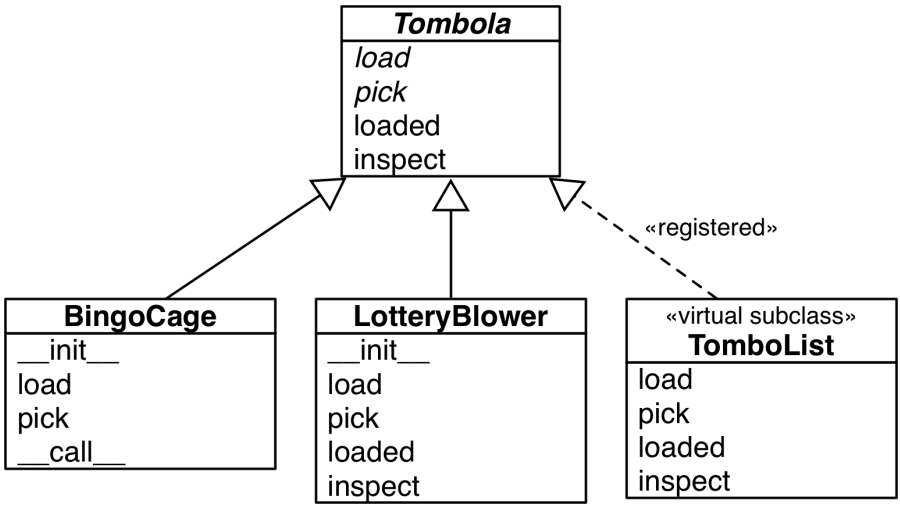


Рис. 13.5. UML-диаграмма ABC и трех его подклассов. Имена класса `Tombola` и его абстрактных методов набраны курсивом в соответствии с соглашениями UML. Пунктирная стрелка обозначает реализацию интерфейса, здесь она показывает, что `TomboList` не только реализует интерфейс `Tombola`, но и зарегистрирован в качестве виртуального подкласса `Tombola`, как мы увидим ниже в этой главе[1]

В примере 13.7 показано определение ABC `Tombola`.

Пример 13.7. tombola.py: `Tombola` – ABC с двумя абстрактными и двумя конкретными методами

```
import abc

class Tombola(abc.ABC): ❶

    @abc.abstractmethod
    def load(self, iterable): ❷
        """Добавить элементы из итерируемого объекта."""

    @abc.abstractmethod
    def pick(self): ❸
        """Удалить случайный элемент и вернуть его.

        Этот метод должен возбуждать исключение `LookupError`, если объект пуст.
        """
```

[1] «registered» и «virtual subclass» – нестандартные термины UML. Мы пользуемся ими, чтобы показать взаимосвязи между классами, специфичные для Python.

```
    def loaded(self): ❹
        """Вернуть `True`, если есть хотя бы 1 элемент, иначе `False`."""
        return bool(self.inspect()) ❺

    def inspect(self):
        """Вернуть отсортированный кортеж, содержащий оставшиеся в данный
        момент элементы.
        """
        items = []
        while True: ❻
            try:
                items.append(self.pick())
            except LookupError:
                break
        self.load(items) ❼
        return tuple(items)
```

❶ Чтобы определить ABC, создаем подкласс `abc.ABC`.
❷ Абстрактный метод помечен декоратором `@abstractmethod`, и зачастую его тело содержит только строку документации[1].
❸ Строка документации сообщает программисту, реализующему метод, что в случае отсутствия элементов нужно возбудить исключение `LookupError`.
❹ ABC может содержать конкретные методы.
❺ Конкретные методы ABC должны зависеть только от открытого интерфейса данного ABC (т. е. от других его конкретных или абстрактных методов или свойств).
❻ Мы не знаем, как в конкретных подклассах будут храниться элементы, но можем построить результат `inspect`, опустошив объект `Tombola` с помощью последовательных обращений к `.pick()`...
❼ ... а затем с помощью `.load(…)` вернуть все элементы обратно.

У абстрактного метода может существовать реализация. Но даже если так, подклассы все равно обязаны переопределить его, однако имеют право вызывать абстрактный метод с помощью функции `super()`, расширяя имеющуюся функциональность, вместо того чтобы реализовывать ее с нуля. Информацию о деталях использования декоратора `@abstractmethod` см. в документации по модулю `abc` (*https://docs.python.org/3/library/abc.html*).

Метод `.inspect()` в примере 13.7, пожалуй, надуманный, но он показывает, что, имея всего лишь методы `.pick()` и `.load(…)`, мы можем узнать, что находится внутри `Tombola`: для этого сначала нужно извлечь все элементы по одному, а затем загрузить их обратно. Мы хотели этим подчеркнуть, что в предоставлении конкретных методов ABC нет ничего плохого, при условии что они зависят только от других методов интерфейса. Конкретные подклассы `Tombola`, знаю-

[1] До появления ABC абстрактные методы обычно возбуждали исключение `NotImplementedError`, показывающее, что за реализацию отвечают подклассы. В Smalltalk-80 тела абстрактных методов вызывали бы метод `subclassResponsibility`, унаследованный от `object`, который породил бы сообщение об ошибке «Мой подкласс должен был бы переопределить одно из моих сообщений».

щие о своих внутренних структурах данных, всегда могут подменить `.inspect()` более эффективной реализацией, но не обязаны это делать.

Метод `.loaded()` в примере 13.7 содержит всего одну строку, но обходится дорого: он строит отсортированный кортеж с помощью `.inspect()` только для того, чтобы применить к нему функцию `bool()`. Этот способ работает, но конкретный класс может поступить гораздо лучше, как мы вскоре увидим.

Отметим, что в нашей «карусельной» реализации `.inspect()` обязательно перехватывать исключение `LookupError`, которое возбуждает метод `self.pick()`. Тот факт, что `self.pick()` может возбуждать исключение `LookupError`, составляет часть его интерфейса, но объявить это в Python можно только в документации (см. строку документации абстрактного метода `pick` в примере 13.7.)

Я выбрал исключение `LookupError` из-за его места в иерархии исключений в Python по отношению к `IndexError` и `KeyError` – исключениям, которые, скорее всего, будут возбуждать операции со структурами данных в конкретных подклассах `Tombola`. Таким образом, реализация может возбуждать исключение `LookupError`, `IndexError`, `KeyError` или пользовательского подкласса `LookupError`. См. рис. 13.6.

```
BaseException
├── SystemExit
├── KeyboardInterrupt
├── GeneratorExit
└── Exception
     ├── StopIteration
     ├── ArithmeticError
     │    ├── FloatingPointError
     │    ├── OverflowError
     │    └── ZeroDivisionError
     ├── AssertionError
     ├── AttributeError
     ├── BufferError
     ├── EOFError
     ├── ImportError
     ├── LookupError  ❶
     │    ├── IndexError  ❷
     │    └── KeyError  ❸
     ├── MemoryError
     ... и т. д.
```

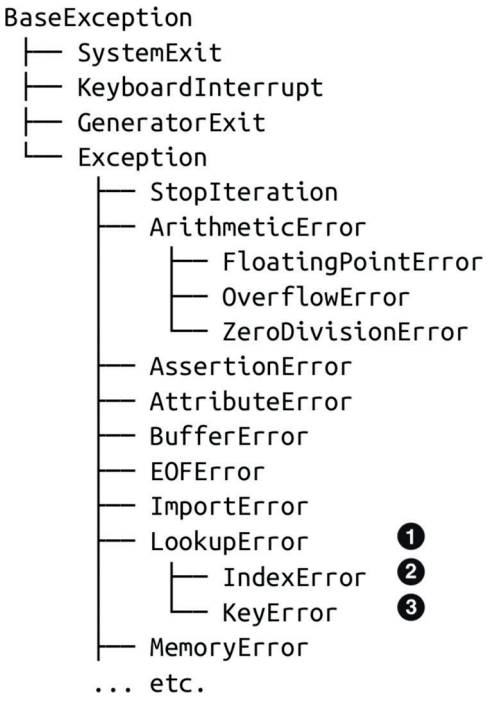


Рис. 13.6. Часть иерархии классов `Exception`[1]

❶ `LookupError` – исключение, обрабатываемое в методе `Tombola.inspect`.

❷ `IndexError` – подкласс `LookupError`, это исключение возбуждается при попытке получить из последовательности элемент с индексом, большим индекса последнего элемента.

❸ Исключение `KeyError` возбуждается при обращении к несуществующему ключу отображения.

Вот мы и создали собственный ABC `Tombola`. Чтобы посмотреть, как производится проверка интерфейса ABC, попробуем обмануть `Tombola`, предоставив дефектную реализацию.

Пример 13.8. Непригодная реализация Tombola не останется незамеченной

```
>>> from tombola import Tombola
>>> class Fake(Tombola): ❶
...     def pick(self):
...         return 13
...
>>> Fake ❷
<class '__main__.Fake'>
>>> f = Fake() ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Fake with abstract method load
```

[1] Полное дерево приведено в разделе 5.4 «Иерархия исключений» справочного руководства по стандартной библиотеке Python.

❶ Объявить `Fake` подклассом `Tombola`.
❷ Класс создан, пока никаких ошибок.
❸ При попытке создать экземпляр класса `Fake` возникает исключение `TypeError`. Сообщение не оставляет сомнений: класс `Fake` считается абстрактным, потому что в нем не реализован метод `load` – один из абстрактных методов, объявленных в ABC `Tombola`.

Итак, мы написали свой первый ABC и проверили, как контролируется его корректность. Скоро мы создадим подкласс `Tombola`, но сначала поговорим о некоторых правилах программирования ABC.

Синтаксические детали ABC

Лучший способ объявить ABC – сделать его подклассом `abc.ABC` или какого-нибудь другого ABC.

Помимо базового класса `ABC` и декоратора `@abstractmethod`, в модуле `abc` определены декораторы `@abstractclassmethod`, `@abstractstaticmethod` и `@abstractproperty`. Однако последние три объявлены нерекомендованными в версии Python 3.3, после того как стало возможно указывать другие декораторы поверх `@abstractmethod`, так что все прочие оказались избыточными. Например, вот как рекомендуется объявлять абстрактный метод класса:

```
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```

Порядок декораторов функции в композиции обычно важен, а в случае `abstractmethod` документация не оставляет никаких сомнений:

> Если abstractmethod() применяется в сочетании с другими дескрипторами метода, он должен быть самым внутренним декоратором...[1]

Иными словами, между `@abstractmethod` и предложением `def` не должно быть никаких других декораторов.

Обсудив синтаксические детали ABC, опробуем `Tombola` на практике, реализовав несколько его конкретных подклассов.

Создание подклассов ABC

Имея ABC `Tombola`, мы теперь разработаем два конкретных подкласса, согласованных с его интерфейсом. Они были изображены на рис. 13.5 вместе с виртуальными подклассами, которые будут рассмотрены в следующем разделе.

Класс `BingoCage` в примере 13.9 – это вариант примера 7.8 с более качественным рандомизатором. В нем реализованы обязательные абстрактные методы `load` и `pick`.

[1] Раздел `@abc.abstractmethod` (*https://docs.python.org/dev/library/abc.html#abc.abstractmethod*) документации по модулю `abc`.

Пример 13.9. bingo.py: `BingoCage` – конкретный подкласс `Tombola`

```
import random

from tombola import Tombola

class BingoCage(Tombola): ❶

    def __init__(self, items):
        self._randomizer = random.SystemRandom() ❷
        self._items = []
        self.load(items) ❸

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items) ❹

    def pick(self): ❺
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self): ❻
        self.pick()
```

❶ Этот класс `BingoCage` явно расширяет `Tombola`.

❷ Качества класса `random.SystemRandom` достаточно для программирования азартных игр в сети, он реализует API `random`, пользуясь функцией `os.urandom(…)`, которая возвращает случайные байты, «пригодные для использования в криптографических приложениях» (документация по модулю `os`, *http://docs.python.org/3/library/os.html#os.urandom*).

❸ Делегировать начальную загрузку методу `.load(…)`.

❹ Вместо функции `random.shuffle()` использовать метод `.shuffle()` нашего экземпляра `SystemRandom`.

❺ Метод `pick` реализован, как в примере 7.8.

❻ Метод `__call__` также заимствован из примера 7.8. Для согласованности с интерфейсом `Tombola` он не нужен, но дополнительные методы никакого вреда не принесут.

`BingoCage` наследует накладный метод `loaded` и простодушный метод `inspect` от `Tombola`. Тот и другой можно переопределить гораздо более быстрыми однострочными методами, как в примере 13.10. Но хочу подчеркнуть: мы можем не утруждать себя и просто унаследовать неоптимальные конкретные методы от ABC. Методы, унаследованные от `Tombola`, работают не так быстро, как могли бы в `BingoCage`, но дают правильные результаты для любого подкласса `Tombola`, в котором корректно реализованы методы `pick` и `load`.

В примере 13.10 показана совершенно другая, но тоже корректная реализация интерфейса `Tombola`. Вместо перетасовывания «шаров» и выталкивания последнего класс `LotteryBlower` выбирает элемент в случайной позиции.

Пример 13.10. lotto.py: `LotteryBlower` – конкретный подкласс, в котором переопределены методы `inspect` и `loaded` ABC `Tombola`

```
import random

from tombola import Tombola

class LottoBlower(Tombola):

    def __init__(self, iterable):
        self._balls = list(iterable) ❶

    def load(self, iterable):
        self._balls.extend(iterable)

    def pick(self):
        try:
            position = random.randrange(len(self._balls)) ❷
        except ValueError:
            raise LookupError('pick from empty LottoBlower')
        return self._balls.pop(position) ❸

    def loaded(self): ❹
        return bool(self._balls)

    def inspect(self): ❺
        return tuple(self._balls)
```

❶ Инициализатор принимает произвольный итерируемый объект, аргумент используется для построения списка.

❷ Функция `random.randrange(...)` возбуждает исключение `ValueError`, если диапазон пуст, мы перехватываем его и возбуждаем взамен исключение `LookupError`, сохраняя совместимость с ABC `Tombola`.

❸ В противном случае из `self._balls` выбирается случайный элемент.

❹ Перегрузить метод `loaded`, чтобы не вызывать `inspect` (как в методе `Tombola.loaded` из примера 13.7). Мы можем ускорить его, работая непосредственно с `self._balls`, – нет необходимости строить заново весь кортеж.

❺ Перегрузить метод `inspect`, новый код состоит всего из одной строки.

В примере 13.10 иллюстрируется достойная отдельного упоминания идиома: в методе `__init__` в атрибуте `self._balls` сохраняется `list(iterable)`, а не просто ссылка на `iterable` (т. е. мы не просто присваиваем `iterable` атрибуту `self._balls`). Как отмечалось в разделе «Защитное программирование и принцип быстрого отказа» выше, это повышает гибкость класса `LotteryBlower`, потому что аргумент `iterable` может быть произвольным итерируемым объектом. Однако элементы из него сохраняются во внутреннем списке, так что нам доступен метод `pop`. И даже если в аргументе `iterable` всегда передается список, вызов `list(iterable)` создает копию аргумента, и это хорошо, поскольку мы удаляем из списка элементы, а клиент может не ожидать, что переданный ему список изменится[1].

[1] Раздел «Защитное программирование при наличии изменяемых параметров» главы 6 посвящен проблеме псевдонимии, которую мы здесь счастливо избежали.

Теперь мы подходим к важнейшей динамической особенности гусиной типизации: объявлению виртуальных подклассов методом `register`.

Виртуальный подкласс Tombola

Важнейшая характеристика гусиной типизации, благодаря которой она и заслужила «водоплавающее» имя, – возможность регистрировать класс как *виртуальный подкласс* ABC, даже без наследования. При этом мы обещаем, что класс честно реализует интерфейс, определенный в ABC, а Python верит нам на слово, не производя проверку. Если мы соврем, то будем наказаны исключением во время выполнения.

Это делается путем вызова метода `register` абстрактного базового класса. В результате зарегистрированный класс становится виртуальным подклассом ABC и распознается в качестве такового функцией `issubclass`, однако не наследует ни методы, ни атрибуты ABC.

Виртуальные подклассы не наследуют ABC, для которых зарегистрированы. Их согласованность с интерфейсом ABC не проверяется никогда, даже в момент создания экземпляра. Кроме того, программы статической проверки типов не могут обработать виртуальные подклассы. Детали см. в проблеме Mypy 2922 «ABCMeta.register support» (*https://github.com/python/mypy/issues/2922*).

Метод `register` чаще всего вызывается как обычная функция (см. раздел «Использование функции register на практике» ниже), но может использоваться и как декоратор. В примере 13.11 мы применяем синтаксис декоратора и реализуем `TomboList`, виртуальный подкласс `Tombola`, изображенный на рис. 13.7.

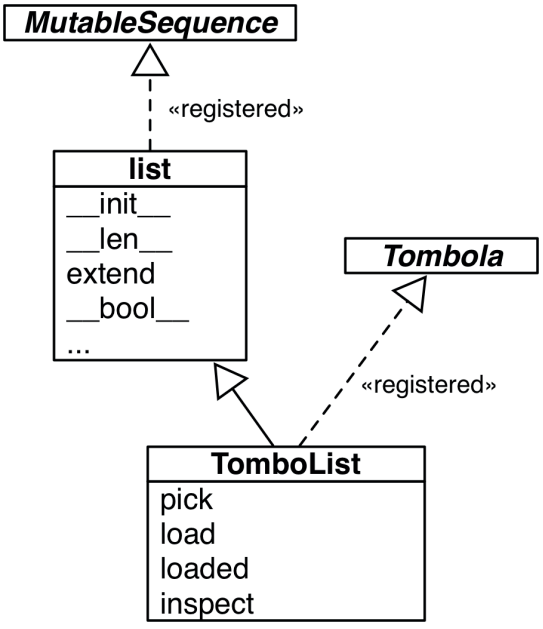


Рис. 13.7. UML-диаграмма классов для `TomboList`, настоящего подкласса `list` и виртуального подкласса `Tombola`

Пример 13.11. tombolist.py: `TomboList` – виртуальный подкласс `Tombola`

```
from random import randrange

from tombola import Tombola

@Tombola.register ❶
class TomboList(list): ❷

    def pick(self):
        if self: ❸
            position = randrange(len(self))
            return self.pop(position) ❹
        else:
            raise LookupError('pop from empty TomboList')

    load = list.extend ❺

    def loaded(self):
        return bool(self) ❻

    def inspect(self):
        return tuple(self)

# Tombola.register(TomboList) ❼
```

❶ `Tombolist` зарегистрирован как виртуальный подкласс `Tombola`.

❷ `Tombolist` расширяет `list`.

❸ `Tombolist` наследует от `list` метод `__bool__`, который возвращает `True`, если список не пуст.

❹ Наш метод `pick` вызывает метод `self.pop`, унаследованный от `list`, передавая ему индекс случайного элемента.

❺ `Tombolist.load` – то же самое, что `list.extend`.

❽ Метод `loaded` делегирует работу методу `bool`[1].

❼ Вызывать метод `register` таким образом можно всегда, и это полезно, когда нужно зарегистрировать класс, который сопровождаете не вы, но который согласован с интерфейсом.

Отметим, что благодаря регистрации функции `issubclass` и `isinstance` считают, что `TomboList` – подкласс `Tombola`:

```
>>> from tombola import Tombola
>>> from tombolist import TomboList
>>> issubclass(TomboList, Tombola)
True
>>> t = TomboList(range(100))
>>> isinstance(t, Tombola)
True
```

[1] Прием, использованный в методе `load()`, для `loaded()` работать не будет, потому что в типе `list` не реализован метод `__bool__`, который я хотел бы связать с `loaded`. С другой стороны, встроенная функция `bool()` не нуждается в методе `__bool__`, потому что может использовать также метод `__len__`. См. раздел 4.1 «Проверка значения истинности» главы «Встроенные типы» документации по Python (*https:// docs.python.org/3/library/stdtypes.html#truth*).

Однако наследование управляется специальным атрибутом класса `__mro__` – Method Resolution Order (порядок разрешения методов). По существу, в нем перечисляются класс и его суперклассы в том порядке, в котором Python просматривает их в поисках методов[1]. Если вывести атрибут `__mro__` класса `TomboList`, то мы увидим в нем только «настоящие» суперклассы – `list` и `object`:

```
>>> TomboList.__mro__
(<class 'tombolist.TomboList'>, <class 'list'>, <class 'object'>)
```

Класса `Tombola` в списке `TomboList.__mro__` нет, поэтому `TomboList` не наследует ни одного метода от `Tombola`.

На этом завершается изучение ABC `Tombola`. В следующем разделе мы рассмотрим, как функция ABC `register` используется на практике.

Использование функции register на практике

В примере 13.11 мы использовали `Tombola.register` в качестве декоратора класса. В версиях, предшествующих Python 3.3, такое использование `register` запрещено – ее следует вызывать как обычную функцию после определения класса (см. комментарий в примере 13.11). Однако, несмотря на то что теперь `register` можно использовать как декоратор, чаще она применяется как функция для регистрации классов, определенных где-то в другом месте. Например, в исходном коде модуля `collections.abc` (*https://github.com/python/cpython/blob/0bbf30e2b910bc9c5899134ae9d73a8df968da35/Lib/_collections_abc.py*) встроенные типы `tuple`, `str`, `range` и `memoryview` зарегистрированы как виртуальные подклассы `Sequence`:

```
Sequence.register(tuple)
Sequence.register(str)
Sequence.register(range)
Sequence.register(memoryview)
```

Еще несколько встроенных типов зарегистрированы как подклассы ABC, содержащихся в файле *_collections_abc.py*. Эти регистрации производятся только при импорте указанного файла, но это нормально, потому что для получения самих ABC модуль так или иначе необходимо импортировать. Например, чтобы выполнить проверку `isinstance(my_dict, MutableMapping)`, необходимо иметь доступ к классу `MutableMapping` из модуля `collections.abc`.

И наследование ABC, и регистрация в качестве виртуального подкласса ABC – явные способы сделать так, чтобы проходили проверки с помощью функции `issubclass`, а равно и проверки с помощью функции `isinstance`, которая опирается на `issubclass`. Но некоторые ABC поддерживают также структурную типизацию. Ниже объясняется, что это такое.

ABC и структурная типизация

ABC чаще всего используются в сочетании с номинальной типизацией. Когда класс `Sub` явно наследует `AnABC` или регистрируется в качестве виртуального подкласса `AnABC`, имя `AnABC` связывается с классом `Sub`, именно поэтому во время выполнения вызов `issubclass(AnABC, Sub)` возвращает `True`.

[1] Ниже целый раздел «Множественное наследование и порядок разрешения методов» посвящен атрибуту класса `__mro__`. А пока нам хватит и краткого объяснения.

Напротив, структурная типизация подразумевает изучение структуры открытого интерфейса объекта с целью определить его тип: объект *совместим с* типом, если он реализует все методы, определенные в типе[1]. Динамическая и статическая утиные типизации – два подхода к структурной типизации.

Оказывается, что некоторые ABC также поддерживают структурную типизацию. В своем эссе «Водоплавающие птицы и ABC» Алекс Мартелли показывает, что класс может быть распознан как виртуальный подкласс ABC даже без регистрации. Приведем еще раз его пример, добавив проверку с использованием функции `issubclass`:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
>>> issubclass(Struggle, abc.Sized)
True
```

Функция `issubclass` (а значит, и `isinstance`) считает класс `Struggle` подклассом `abc.Sized`, потому что `abc.Sized` реализует специальный метод класса `__subclasshook__` .

Метод `__subclasshook__` в классе `Sized` проверяет, имеет ли переданный в аргументе класс атрибут с именем `__len__`. Если да, то класс считается виртуальным подклассом `Sized`. См. пример 13.12.

Пример 13.12. Определение класса `Sized` из файла Lib/_collections_abc.py (*https://github.com/python/cpython/blob/0fbddb14dc03f61738af01af88e7d8aa8df07336/Lib/_collections_abc.py#L369*)

```
class Sized(metaclass=ABCMeta):

    __slots__ = ()

    @abstractmethod
    def __len__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Sized:
            if any("__len__" in B.__dict__ for B in C.__mro__): ❶
                return True ❷
        return NotImplemented ❸
```

❶ Если в словаре `__dict__` любого класса, перечисленного в `C.__mro__` (т. е. C и его суперклассах), существует атрибут с именем `__len__` …

❷ … то вернуть `True`, сигнализируя о том, что `C` – виртуальный подкласс `Sized`.

❸ Иначе вернуть `NotImplemented`, чтобы продолжить проверку подкласса.

[1] Концепция совместимости типов объяснялась в разделе «“Является подтипом” и “совместим с”» главы 8.

Если вас интересуют детали проверки подкласса, загляните в исходный код метода `ABCMeta.__subclasscheck__` в файле *Lib/abc.py* (*https://github.com/python/cpython/ blob/c0a9afe2ac1820409e6173bd1893ebee2cf50270/Lib/abc.py#L196*). Предупреждение: в этом коде уйма if'ов и два рекурсивных вызова. В Python 3.7 Иван Левкивский и Инада Наоки переписали на C большую часть модуля `abc`, чтобы повысить производительность. См. проблему Python #31333 (*https://bugs.python.org/issue31333*). Текущая реализация `ABCMeta.__subclasscheck__` просто вызывает `_abc_subclasscheck`. Относящийся к делу исходный код на C находится в файле *cpython/Modules/_abc.c#L605* (*https://github.com/python/cpython/blob/3635388f52b42e5280229104747962117104c453/Modules/_abc.c#L605*).

Вот так метод `__subclasshook__` позволяет ABC поддержать структурную типизацию. Несмотря на наличие формального определения интерфейса в ABC и скрупулезных проверок, осуществляемых функцией `isinstance`, в определенных контекстах вполне можно использовать никак не связанный с ABC класс просто потому, что в нем реализован определенный метод (или потому, что он постарался убедить `__subclasshook__`, что за него можно поручиться).

Следует ли реализовывать `__subclasshook__` в своих собственных ABC? Пожалуй, нет. Все реализации `__subclasshook__`, которые я встречал в исходном коде Python, находятся в ABC типа `Sized`, где объявлен только один специальный метод, и они просто проверяют имя этого метода. Учитывая «специальный» статус таких методов, можно с некоторой долей уверенности предположить, что любой метод с именем `__len__` делает именно то, чего вы от него ожидаете. Но, даже не выходя за пределы специальных методов и фундаментальных ABC, делать такие предположения рискованно. Например, все отображения реализуют методы `__len__`, `__getitem__` и `__iter__`, но они справедливо не считаются подтипами `Sequence`, поскольку не позволяют получить элемент по целочисленному смещению или срезу. Именно поэтому класс `abc.Sequence` не реализует метод `__subclasshook__`.

Для тех же ABC, которые могли бы написать вы или я, полагаться на метод `__subclasshook__` еще более рискованно. Лично я не готов поверить, что класс с именем `Spam`, который реализует или наследует методы `load`, `pick`, `inspect` и `loaded`, гарантированно ведет себя как `Tombola`. Пусть уж лучше программист явно подтвердит это, сделав `Spam` подклассом `Tombola` или хотя бы зарегистрировав его: `Tombola.register(Spam)`. Конечно, ваш метод `__subclasshook__` мог бы еще проверить сигнатуры методов и другие свойства, но не думаю, что оно того стоит.

Статические протоколы

Впервые речь о статических протоколах заходила в разделе «Статические протоколы» главы 8. Я подумывал о том, чтобы вообще отложить рассмотрение протоколов до этой главы, но потом решил, что введение в аннотации типов в функциях не может обойтись без протоколов, потому утиная типизация – неотъемлемая часть Python, а статическая проверка типов без протоколов не очень хорошо справляется с питоническими API.

Мы завершим эту главу иллюстрацией статических протоколов на двух простых примерах, а также обсудим числовые ABC и протоколы. Для начала покажем, как статический протокол позволяет аннотировать и проверить тип функции `double()`, которую мы уже встречали в разделе «Типы определяются тем, какие операции они поддерживают» главы 8.

Типизированная функция double

Рассказывая о Python программистам, привыкшим к статически типизированным языкам, я люблю приводить в пример следующую простую функцию `double`:

```
>>> def double(x):
...     return x * 2
...
>>> double(1.5)
3.0

>>> double('A')
'AA'
>>> double([10, 20, 30])
[10, 20, 30, 10, 20, 30]
>>> from fractions import Fraction
>>> double(Fraction(2, 5))
Fraction(4, 5)
```

До появления статических протоколов не существовало способа добавить аннотации типов к `double`, не ограничивая возможностей ее применения[1].

Благодаря утиной типизации `double` будет работать даже с типами, которых пока не существует, например с улучшенным классом `Vector`, с которым мы познакомимся в разделе «Перегрузка оператора * для скалярного умножения» (глава 16).

```
>>> from vector_v7 import Vector
>>> double(Vector([11.0, 12.0, 13.0]))
Vector([22.0, 24.0, 26.0])
```

Первоначальная реализация аннотаций типов в Python была основана на номинальной системе типов: имя типа в аннотации должно было совпадать с именем типа фактического аргумента или с именем одного из его суперклассов. Поскольку невозможно поименовать все типы, которые реализуют протокол путем поддержки требуемых операций, утиную типизацию нельзя было описать с помощью аннотаций типов до версии Python 3.8.

Теперь благодаря классу `typing.Protocol` мы можем сообщить Муру, что `double` принимает аргумент `x`, поддерживающий операцию `x * 2`. В примере 13.13 показано, как именно.

[1] Ну хорошо, `double()` полезна разве что в качестве примера. Но в стандартной библиотеке Python много функций, которые нельзя было правильно аннотировать до включения статических протоколов в Python 3.8. Я помогал исправлять пару ошибок в *typeshed* путем добавления аннотаций типов с применением протоколов. Например, в запросе на включение, который исправлял ошибку «Должна ли Муру предупреждать о потенциально недопустимых аргументах max?» (*https://github.com/python/typeshed/issues/4051*) был задействован протокол `_SupportsLessThan`, который я использовал для улучшения аннотаций `max`, `min`, `sorted` и `list.sort`.

Пример 13.13. double_protocol.py: определение `double` с использованием `Protocol`

```
from typing import TypeVar, Protocol

T = TypeVar('T') ❶

class Repeatable(Protocol):
    def __mul__(self: T, repeat_count: int) -> T: ... ❷

RT = TypeVar('RT', bound=Repeatable) ❸

def double(x: RT) -> RT: ❹
    return x * 2
```

❶ Мы будем использовать этот тип `T` в сигнатуре `__mul__`.

❷ `__mul__` – существо протокола `Repeatable`. Параметр `self` обычно не аннотируется – предполагается, что его тип – сам класс. Здесь мы используем `T`, чтобы тип результата гарантированно совпадал с типом `self`. Отметим также, что в этом протоколе `repeat_count` может иметь только тип `int`.

❸ Протокол `Repeatable` связывает переменную-тип `RT`: средство проверки типов потребует, чтобы фактический тип реализовывал `Repeatable`.

❹ Теперь средство проверки типов может проверить, что параметр `x` является объектом, который можно умножать на целое число, и при этом получается значение того же типа, что и `x`.

Этот пример объясняет, почему документ PEP 544 (*https://peps.python.org/pep-0544/*) назван «Protocols: Structural subtyping (static duck typing)» (Протоколы: структурная подтипизация (статическая утиная типизация)). Номинальный тип фактического аргумента `x`, переданного `double`, не играет роли, коль скоро он умеет квакать – т. е. реализует метод `__mul__`.

Статические протоколы, допускающие проверку во время выполнения

На карте типизации (рис. 13.1) `typing.Protocol` располагается в области статической проверки – в нижней половине диаграммы. Но при определении подкласса `typing.Protocol` мы можем использовать декоратор `@runtime_checkable`, чтобы протокол поддерживал проверки с помощью функций `isinstance/issubclass` во время выполнения. Это работает, потому что `typing.Protocol` – абстрактный базовый класс, а значит, поддерживает метод `__subclasshook__`, с которым мы встречались в разделе «Структурная типизация с помощью ABC» выше.

В версии Python 3.9 модуль `typing` включает семь готовых к использованию протоколов, допускающих проверку во время выполнения. Мы упомянем два из них, процитировав описание из документации по модулю `typing` (*https://docs.python.org/3/library/typing.html#protocols*):

`class typing.SupportsComplex`

ABC с одним абстрактным методом, `__complex__`.

`class typing.SupportsFloat`

ABC с одним абстрактным методом, `__float__`.

Эти протоколы предназначены для проверки числовых типов на «конвертируемость»: если объект `o` реализует метод `__complex__`, то вызов `complex(o)` должен вернуть число типа `complex`, потому что специальный метод `__complex__` как раз и существует для поддержки встроенной функции `complex()`.

В примере 13.14 приведен исходный код протокола `typing.SupportsComplex`.

Пример 13.14. Исходный код протокола `typing.SupportsComplex`

```
@runtime_checkable
class SupportsComplex(Protocol):
    """ABC с одним абстрактным методом __complex__."""
    __slots__ = ()

    @abstractmethod
    def __complex__(self) -> complex:
        pass
```

Ключом является абстрактный метод `__complex__`[1]. В процессе статической проверки типов объект будет считаться *совместимым с* протоколом `SupportsComplex`, если он реализует метод `__complex__`, принимающий только аргумент `self` и возвращающий `complex`.

Благодаря применению декоратора класса `@runtime_checkable` к `SupportsComplex` этот протокол теперь можно использовать в сочетании с функцией `isinstance`, как показано в примере 13.15.

Пример 13.15. Использование `SupportsComplex` во время выполнения

```
>>> from typing import SupportsComplex
>>> import numpy as np
>>> c64 = np.complex64(3+4j) ❶
>>> isinstance(c64, complex) ❷
False
>>> isinstance(c64, SupportsComplex) ❸
True
>>> c = complex(c64) ❹
>>> c
(3+4j)
>>> isinstance(c, SupportsComplex) ❺
False
>>> complex(c)
(3+4j)
```

❶ `complex64` – один из пяти комплексных числовых типов в NumPy.

❷ Ни один из комплексных типов NumPy не является подклассом встроенного типа `complex`.

❸ Но все они реализуют метод `__complex__`, поэтому согласованы с протоколом `SupportsComplex`.

❹ Поэтому мы можем создавать их них объекты встроенного типа `complex`.

❺ К сожалению, встроенный тип `complex` не реализует метод `__complex__`, хотя `complex(c)` успешно работает, если `c` имеет тип `complex`.

[1] Атрибут `__slots__` к текущему обсуждению не имеет отношения – это оптимизация, которая была рассмотрена в разделе «Экономия памяти с помощью атрибута класса `__slots__`» главы 11.

Последний пункт, в частности, означает, что если мы хотим проверить, верно ли, что объект `c` имеет тип `complex` или `SupportsComplex`, то можем предоставить кортеж типов в качестве второго аргумента `isinstance`:

```
isinstance(c, (complex, SupportsComplex))
```

Альтернатива – воспользоваться ABC `Complex`, определенным в модуле `numbers`. Встроенный тип `complex`, а также типы NumPy `complex64` и `complex128` зарегистрированы как виртуальные подклассы `numbers.Complex`, поэтому следующий код работает:

```
>>> import numbers
>>> isinstance(c, numbers.Complex)
True
>>> isinstance(c64, numbers.Complex)
True
```

В первом издании книги я рекомендовал использовать абстрактные базовые классы из пакета `numbers`, но теперь мой совет утратил актуальность, потому что эти ABC не распознаются средствами статической проверки типов, как мы увидим в разделе «ABC из пакета numbers и числовые протоколы».

В этом разделе я хотел продемонстрировать, что протокол, допускающий проверку во время выполнения, работает с функцией `isinstance`, но, как объяснено во врезке «Утиная типизация – ваш друг», это не особенно хороший пример использования `isinstance`.

Если вы пользуетесь внешним средством проверки типов, то явная проверка с помощью `isinstance` дает одно преимущество: встретив предложение `if` с условием вида `isinstance(o, MyType)`, Муру может сделать вывод, что внутри блока `if` тип объекта `o` *совместим с* `MyType`.

Утиная типизация – ваш друг

Очень часто бывает, что во время выполнения утиная типизация – лучший подход к проверке типа: вместо того чтобы вызывать `isinstance` или `hasattr`, просто попробуйте выполнить над объектом нужную операцию и обработайте исключения. Приведем конкретный пример.

Вслед предыдущему обсуждению – дан объект `o`, который я хочу использовать как комплексное число, – можно подойти к решению следующим образом:

```
if isinstance(o, (complex, SupportsComplex)):
    # сделать что-то, требующее, чтобы `o` можно было преобразовать в complex
else:
    raise TypeError('o must be convertible to complex')
```

Гусиная типизация подразумевала бы использование ABC `numbers.Complex`:

```
if isinstance(o, numbers.Complex):
    # сделать что-то с `o`, экземпляром класса `Complex`
else:
    raise TypeError('o must be an instance of Complex')
```

Но я предпочитаю использовать утиную типизацию и принцип EAFP (it's easier to ask for forgiveness than permission) – проще попросить прощения, чем испрашивать разрешения):

```
try:
    c = complex(o)
except TypeError as exc:
    raise TypeError('o must be convertible to complex') from exc
```

А если вы в любом случае собирались возбудить исключение `TypeError`, то можно опустить предложения `try/except/raise` и просто написать:

```
    c = complex(o)
```

В этом последнем случае, если `o` – недопустимый тип, Python возбудит исключение с предельно ясным сообщением. Например, вот что я получу, если `o` имеет тип `tuple`:

```
    TypeError: complex() first argument must be a string or a number, not 'tuple'
```

Мне кажется, что подход на основе утиной типизации в этом случае гораздо лучше.

Итак, мы знаем, как использовать статические протоколы во время выполнения с такими уже имеющимися типами, как `complex` и `numpy.complex64`. Теперь обсудим ограничения протоколов, допускающих проверку во время выполнения.

Ограничения протоколов, допускающих проверку во время выполнения

Мы видели, что в общем случае аннотации типов игнорируются во время выполнения, и это также влияет на проверки статических протоколов с помощью функций `isinstance` и `issubclass`.

Например, любой класс, имеющий метод `__float__`, считается – на этапе выполнения – виртуальным подклассом `SupportsFloat`, даже если метод `__float__` не возвращает значение типа `float`.

Рассмотрим следующий консольный сеанс:

```
>>> import sys
>>> sys.version
'3.9.5 (v3.9.5:0a7dcbdb13, May 3 2021, 13:17:02) \n[Clang 6.0
(clang-600.0.57)]'
>>> c = 3+4j
>>> c.__float__
<method-wrapper '__float__' of complex object at 0x10a16c590>
>>> c.__float__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
```

В Python 3.9 тип `complex` содержит метод `__float__`, но лишь для того, чтобы возбудить исключение `TypeError` с понятным сообщением об ошибке. Если бы у метода `__float__` были аннотации, то тип возвращаемого значения был бы `NoReturn` – как мы видели в разделе «Тип NoReturn» главы 8.

Но аннотирование типа `complex.__float__` в *typeshed* не решило бы этой проблемы, потому что среда выполнения Python, вообще говоря, игнорирует аннотации типов – да и вообще не имеет доступа к файлам заглушек в *typeshed*.

Продолжим предыдущий сеанс в Python 3.9:

```
>>> from typing import SupportsFloat
>>> c = 3+4j
>>> isinstance(c, SupportsFloat)
True
>>> issubclass(complex, SupportsFloat)
True
```

Мы имеем обескураживающий результат: проверка `SupportsFloat` во время выполнения говорит, что `complex` можно преобразовать в `float`, а на самом деле мы получаем ошибку типизации.

Конкретная ошибка для типа `complex` была исправлена в Python 3.10.0b4 просто удалением метода `complex.__float__`.

Но общая проблема осталась: функции `isinstance/issubclass` смотрят только на наличие или отсутствие методов, не проверяя их сигнатуры, а уж тем более аннотации типов. И такое положение дел не изменится, потому что подобные проверки типа во время выполнения повлекли бы за собой неприемлемые затраты[1].

Теперь посмотрим, как реализовать статический протокол в пользовательском классе.

Поддержка статического протокола

Вспомним класс `Vector2d`, разработанный в главе 11. Учитывая, что и комплексное число, и экземпляр `Vector2d` состоят из двух чисел с плавающей точкой, имеет смысл поддержать преобразование `Vector2d` в `complex`.

В примере 13.16 приведена реализация метода `__complex__`, дополняющего последнюю версию `Vector2d` из примера 11.11. Для полноты можно поддержать и обратную операцию с помощью метода класса `fromcomplex`, который строит экземпляр `Vector2d` из `complex`.

Пример 13.16. vector2d_v4.py: метод преобразования в `complex` и обратно

```
    def __complex__(self):
        return complex(self.x, self.y)

    @classmethod
    def fromcomplex(cls, datum):
        return cls(datum.real, datum.imag) ❶
```

❶ Предполагается, что у `datum` есть атрибуты `.real` и `.imag`. Улучшенная реализация будет приведена в примере 13.17.

Имея этот код и метод `__abs__`, который уже был в версии `Vector2d` из примера 11.11, мы получаем следующие свойства:

[1] Спасибо Ивану Левкивскому, соавтору документа PEP 544 (о протоколах), который указал, что проверка типа не сводится к проверке того, что типом `x` является `T`: проверяется, что тип `x` *совместим с* `T`, а это может быть дорого. Неудивительно, что Муру требуется несколько секунд для проверки даже короткого Python-скрипта.

```
>>> from typing import SupportsComplex, SupportsAbs
>>> from vector2d_v4 import Vector2d
>>> v = Vector2d(3, 4)
>>> isinstance(v, SupportsComplex)
True
>>> isinstance(v, SupportsAbs)
True
>>> complex(v)
(3+4j)
>>> abs(v)
5.0
>>> Vector2d.fromcomplex(3+4j)
Vector2d(3.0, 4.0)
```

С точки зрения проверки типа во время выполнения, пример 13.16 годится, но для более полного покрытия программами статической проверки типов и более понятных сообщений об ошибках со стороны Муру методы `__abs__`, `__complex__` и `fromcomplex` должны иметь аннотации типов, как показано в примере 13.17.

Пример 13.17. vector2d_v5.py: добавление аннотаций в интересующие нас методы

```
def __abs__(self) -> float: ❶
    return math.hypot(self.x, self.y)

def __complex__(self) -> complex: ❷
    return complex(self.x, self.y)

@classmethod
def fromcomplex(cls, datum: SupportsComplex) -> Vector2d: ❸
    c = complex(datum) ❹
    return cls(c.real, c.imag)
```

❶ Аннотация типа возвращаемого значения `float` необходима, иначе Муру выведет тип `Any` и не будет проверять тело метода.

❷ Даже без аннотации Муру сумела вывести, что этот метод возвращает `complex`. Наличие аннотации предотвратит выдачу предупреждения (в зависимости от конфигурации Муру).

❸ Здесь наличие `SupportsComplex` гарантирует, что `datum` допускает преобразование.

❹ Это явное преобразование необходимо, потому что `SupportsComplex` не объявляет атрибуты `.real` и `.imag`, используемые в следующей строке. Например, у `Vector2d` нет этих атрибутов, но он реализует метод `__complex__`.

Тип возвращаемого `fromcomplex` значения может быть `Vector2d`, если в начале модуля находится предложение `from __future__ import annotations`. Этот импорт приводит к тому, что аннотации типов сохраняются в виде строк, а не вычисляются на этапе импорта, когда интерпретатор обрабатывает определения функций. Если бы `__future__` не импортировался из `annotations`, то ссылка `Vector2d` была бы в этой точке недопустима (класс еще не полностью определен) и ее следовало бы записать в виде строки: `'Vector2d'`, как если бы это была опережающая ссылка. Импорт `__future__` был предложен в документе PEP 563 «Postponed Evaluation of Annotations» (*https://www.python.org/dev/peps/pep-0563/*) и реализован в Python 3.7. Предполагалось, что это поведение станет стандартным в вер-

сии 3.10, но потом изменение отложили до следующей версии[1]. После того как это произойдет, импорт станет излишним, но и вреда не принесет.

Теперь посмотрим, как создавать, а затем и расширять новый статический протокол.

Проектирование статического протокола

Изучая гусиную типизацию, мы видели ABC `Tombola` в разделе «Определение и использование ABC». А сейчас покажем, как определить аналогичный интерфейс с помощью статического протокола.

В ABC `Tombola` определено два метода: `pick` и `load`. Мы могли бы определить статический протокол с такими же двумя методами, но от сообщества пользователей языка Go я узнал, что протоколы с единственным методом делают статическую утиную типизацию более полезной и гибкой. В стандартной библиотеке Go есть несколько интерфейсов, например `Reader` – интерфейс ввода-вывода, который требует только метода `read`. Если спустя некоторое время вы поймете, что необходим более полный протокол, то сможете определить его, объединив два или более протоколов.

Использование контейнера, который случайным образом выбирает элементы, может потребовать перезагрузки контейнера, а может и не потребовать. Но оно, безусловно, требует метода для самого выбора, поэтому именно такой метод я буду считать необходимой принадлежностью минимального протокола `RandomPicker`. Код этого протокола приведен в примере 13.18, а его использование продемонстрировано на тестах из примера 13.19.

Пример 13.18. randompick.py: определение `RandomPicker`

```
from typing import Protocol, runtime_checkable, Any

@runtime_checkable
class RandomPicker(Protocol):
    def pick(self) -> Any: ...
```

Метод `pick` возвращает тип `Any`. В разделе «Реализация обобщенного статического протокола» мы увидим, как сделать `RandomPicker` обобщенным типом с параметром, который позволяет пользователям протокола задавать тип значения, возвращаемого методом `pick`.

Пример 13.19. randompick_test.py: использование `RandomPicker`

```
import random
from typing import Any, Iterable, TYPE_CHECKING

from randompick import RandomPicker ❶

class SimplePicker: ❷
    def __init__(self, items: Iterable) -> None:
        self._items = list(items)
        random.shuffle(self._items)
```

[1] Читайте решение Руководящего комитета на *python-dev* (*https://mail.python.org/archives/list/python-dev@python.org/thread/CLVXXPQ2T2LQ5MP2Y53VVQFCXYWQJHKZ/*).

```
    def pick(self) -> Any: ❸
        return self._items.pop()

def test_isinstance() -> None: ❹
    popper: RandomPicker = SimplePicker([1]) ❺
    assert isinstance(popper, RandomPicker) ❻

def test_item_type() -> None: ❼
    items = [1, 2]
    popper = SimplePicker(items)
    item = popper.pick()
    assert item in items
    if TYPE_CHECKING:
        reveal_type(item) ❽
    assert isinstance(item,int)
```

❶ Необязательно импортировать статический протокол, чтобы определить класс, который его реализует. Здесь я импортировал `RandomPicker`, только чтобы использовать `test_isintance` ниже.

❷ `SimplePicker` реализует `RandomPicker`, но не является его подклассом. Это статическая утиная типизация в действии.

❸ `Any` – тип возвращаемого значения по умолчанию, так что эта аннотация, строго говоря, лишняя, но она проясняет наше намерение реализовать протокол `RandomPicker`, определенный в примере 13.18.

❹ Не забывайте добавлять аннотации `-> None` в свои тесты, если хотите, чтобы Муру обращала на них внимание.

❺ Я добавил аннотацию типа для переменной `popper`, чтобы показать, что Муру понимает, что `SimplePicker` *совместим с*.

❻ Этот тест доказывает, что экземпляр `SimplePicker` является также экземпляром `RandomPicker`. Это работает, потому что к `RandomPicker` применен декоратор `@runtime_checkable`, а `SimplePicker` имеет обязательный метод `pick`.

❼ Этот тест вызывает метод `pick` класса `SimplePicker`, проверяет, что он возвращает один из элементов, переданных `SimplePicker`, а затем производит статические и динамические проверки возвращенного элемента.

❽ Эта строка генерирует примечание в выводе Муру.

Как мы видели в примере 8.22, `reveal_type` – «магическая» функция, распознаваемая Муру, поэтому она не импортируется, но вызывать ее можно только внутри блоков `if` с условием `typing.TYPE_CHECKING`, которое средство статической проверки типов считает равным `True`, а среда выполнения – `False`.

Оба теста в примере 13.19 проходят. Муру тоже не находит ошибок в этом коде и показывает результат `reveal_type` для объекта `item`, возвращенного `pick`:

```
$ mypy randompick_test.py
randompick_test.py:24: note: Revealed type is 'Any'
```

Создав свой первый статический протокол, давайте познакомимся с тем, что советуют по этому поводу знающие люди.

Рекомендации по проектированию протоколов

После 10 лет опыта работы со статической утиной типизацией в Go стало ясно, что узкие протоколы более полезны. Часто в таких протоколах есть всего один

метод, а более двух – редкость. Мартин Фаулер написал статью, в которой определяется *ролевой интерфейс* (*https://martinfowler.com/bliki/RoleInterface.html*), эту идею полезно иметь в виду при проектировании протоколов.

Кроме того, иногда можно увидеть, как протокол определяется рядом с использующей его функцией, т. е. в «клиентском коде», а не в библиотеке. Это упрощает создание новых типов, вызывающих данную функцию, что неплохо с точки зрения расширяемости и тестирования с помощью объектов-имитаций (mock).

Использование узких протоколов и протоколов в клиентском коде позволяет избежать ненужной тесной сцепленности, в полном соответствии с принципом разделения интерфейса (*https://en.wikipedia.org/wiki/Interface_segregation_ principle*), который можно кратко формулировать так: «Клиента не следует принуждать к зависимости от интерфейсов, которыми он не пользуется».

На странице «Как предлагать свой вклад в typeshed» (*https://github.com/python/typeshed/blob/master/CONTRIBUTING.md*) рекомендуется следующее соглашение об именовании:

- выбирайте для протоколов простые имена, которые ясно описывают концепцию (например, `Iterator`, `Container`);
- используйте имя вида `SupportsX` для протоколов, предоставляющих вызываемые методы (например, `SupportsInt`, `SupportsRead`, `SupportsReadSeek`)[1];
- используйте имя вида `HasX` для протоколов, которые имеют атрибуты, допускающие чтение и (или) запись, либо методы чтения и установки (например, `HasItems`, `HasFileno`).

В стандартной библиотеке Go принято соглашение об именовании, которое мне нравится: для протоколов с единственным методом, если имя метода – глагол, то добавляйте суффикс «-er» или «-or», чтобы сделать имя протокола существительным. Например, называйте не `SupportsRead`, а `Reader`. Еще примеры: `Formatter`, `Animator`, `Scanner`. В поисках источника вдохновения прочитайте статью Асука Кендзи «Go (Golang) Standard Library Interfaces (Selected)» (*https://gist.github.com/asukakenji/ ac8a05644a2e98f1d5ea8c299541fce9*).

Веская причина создавать минималистские протоколы – возможность их последующего расширения при необходимости. Сейчас мы покажем, как просто создать производный протокол, содержащий дополнительный метод.

Расширение протокола

В начале предыдущего раздела я говорил, что разработчики Go ратуют за минимализм при определении интерфейсов – так они называют статические протоколы. Многие широко используемые в Go интерфейсы имеют всего один метод.

Если практика показывает, что протокол с большим числом методов полезен, то лучше не добавлять методы в исходный протокол, а произвести от него новый. У расширения статического протокола в Python есть несколько подводных камней, как показывает пример 13.20.

[1] Любой метод является вызываемым, так что эта рекомендация мало что дает. Быть может, имелось в виду «предоставляющих один или два метода»? Так или иначе, это рекомендация, а не непреложное правило.

Пример 13.20. randompickload.py: расширение `RandomPicker`

```
from typing import Protocol, runtime_checkable
from randompick import RandomPicker

@runtime_checkable ❶
class LoadableRandomPicker(RandomPicker, Protocol): ❷
    def load(self, Iterable) -> None: ... ❸
```

❶ Если нужно, чтобы производный протокол допускал проверку во время выполнения, то необходимо повторно применить к нему декоратор – это поведение не наследуется[1].

❷ Для любого протокола необходимо явно указать `typing.Protocol` в качестве одного из базовых классов, в дополнение к расширяемому протоколу. Обычное наследование в Python работает не так[2].

❸ Возвращаемся к «регулярному» ООП: нужно объявлять только новый метод производного протокола. Объявление метода `pick` унаследовано от `RandomPicker`.

На этом завершается последний пример определения и использования статического протокола в этой главе.

В заключение рассмотрим числовые ABC и их возможную замену числовыми протоколами.

ABC из пакета numbers и числовые протоколы

В разделе «Падение числовой башни» главы 8 мы видели, что ABC из пакета `numbers` стандартной библиотеки прекрасно работают в сочетании с проверкой типов во время выполнения.

Если нужно проверить, принадлежит ли объект целочисленному типу, можно воспользоваться вызовом `isinstance(x,numbers.Integral)`, который возвращает `True` для `int`, `bool` (подкласс `int`) и других целочисленных типов, предоставляемых внешними библиотеками, которые зарегистрировали свои типы как виртуальные подклассы ABC из пакета `numbers`. Например, в NumPy имеется 21 целый тип (*https://numpy.org/devdocs/user/basics.types.html*), а также несколько типов с плавающей точкой, зарегистрированных как `numbers.Real`, и комплексных типов с различной разрядностью, зарегистрированных как `numbers.Complex`.

Удивительно, но тип `decimal.Decimal` не зарегистрирован как виртуальный подкласс `numbers.Real`. Причина в том, что если в программе нужна точность `Decimal`, то, наверное, вы предпочли бы застраховаться от случайного смешения десятичных чисел и менее точных чисел с плавающей точкой.

---

[1] Детали и обоснование см. в разделе о декораторе `@runtime_checkable` документа PEP 544 «Protocols: Structural subtyping (static duck typing)» (*https://peps.python.org/pep-0544/#runtime-checkable-decorator-and-narrowing-types-by-isinstance*).

[2] И снова отсылаю читателя, жаждущего деталей и обоснования, к документу PER 544, но на этот раз к разделу «Merging and extending protocols» (*https://peps.python.org/pep-0544/#merging-and-extending-protocols*).

К сожалению, числовая башня не предназначалась для статической проверки типов. Корневой ABC – `numbers.Number` – не имеет методов, поэтому если объявить `x: Number`, то Муру не позволит производить арифметические операции с `x` или вызывать какие-то методы `x`.

Но если ABC из пакета `numbers` не поддерживаются, то какие имеются альтернативы?

Начинать поиск решений проблем, относящихся к типизации, имеет смысл с проекта *typeshed*. Модуль `statistics`, входящий в стандартную библиотеку Python, сопровождается содержащим аннотации типов файлом заглушки *statistics.pyi* (*https://github.com/python/typeshed/blob/master/stdlib/statistics.pyi*) в *typeshed*. Там вы найдете следующие определения, которые используются для аннотирования нескольких функций:

```
_Number = Union[float, Decimal, Fraction]
_NumberT = TypeVar('_NumberT', float, Decimal, Fraction)
```

Этот подход корректный, но имеет ограничения. Он не поддерживает числовые типы за пределами стандартной библиотеки, в отличие от ABC из пакета `numbers` – при условии что числовые типы зарегистрированы как виртуальные подклассы.

В настоящее время рекомендуется использовать числовые протоколы, предоставляемые модулем `typing`, как обсуждалось в разделе «Статические протоколы, допускающие проверку во время выполнения».

К сожалению, во время выполнения числовые протоколы могут вас разочаровать. Как упоминалось в разделе «Ограничения протоколов, допускающих проверку во время выполнения», тип `complex` в Python 3.9 реализует метод `__float__`, но только для того, чтобы возбудить исключение `TypeError` с недвусмысленным сообщением: «can't convert complex to float» (не могу преобразовать complex в float). По той же причине он реализует метод `__int__`. Из-за наличия этих методов `isinstance` возвращает вводящие в заблуждение результаты в Python 3.9. В Python 3.10 методы `complex`, которые безусловно возбуждали исключение `TypeError`, были удалены[1].

С другой стороны, комплексные типы в NumPy реализуют методы `__float__` и `__int__`, которые работают, но выдают предупреждение при первом использовании:

```
>>> import numpy as np
>>> cd = np.cdouble(3+4j)
>>> cd
(3+4j)
>>> float(cd)
<stdin>:1: ComplexWarning: Casting complex values to real
discards the imaginary part
3.0
```

Иногда встречается и противоположная проблема: встроенные типы `complex`, `float` и `int`, а также типы `numpy.float16` и `numpy.uint8` не имеют метода `__complex__`, по-

[1] См. проблему #41974 – Remove complex.__float__, complex.__floordiv__, etc. (*https://bugs.python.org/issue41974*).

этому `isinstance(x, SupportsComplex)` возвращает для них `False`[1]. Комплексные типы NumPy, в частности `np.complex64`, реализуют `__complex__` для преобразования во встроенный тип `complex`.

Однако на практике встроенный конструктор `complex()` обрабатывает экземпляры всех этих типов, не выдавая ни ошибок, ни предупреждений:

```
>>> import numpy as np
>>> from typing import SupportsComplex
>>> sample = [1+0j, np.complex64(1+0j), 1.0, np.float16(1.0), 1,
np.uint8(1)]
>>> [isinstance(x, SupportsComplex) for x in sample]
[False, True, False, False, False, False]
>>> [complex(x) for x in sample]
[(1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j)]
```

Это показывает, что функция `isinstance` проверяет наличие метода `SupportsComplex`, и его отсутствие должно было бы привести к ошибке преобразования в `complex`, но в действительности все преобразования выполняются. В списке рассылки typing-sig Гвидо ван Россум указал, что встроенный конструктор `complex` принимает единственный аргумент, и по этой причине указанные преобразования работают.

С другой стороны, Муру принимает аргументы всех шести типов при обращении к следующей функции `to_complex()`:

```
def to_complex(n: SupportsComplex) -> complex:
    return complex(n)
```

На момент написания книги в NumPy не было аннотаций типов, поэтому все числовые типы из этой библиотеки распознаются как `Any`[2]. С другой стороны, Муру каким-то образом «знает», что встроенные типы `int` и `float` можно преобразовать в `complex`, хотя в *typeshed* только встроенный класс `complex` имеет метод `__complex__`[3].

Итак, хотя проверка числовых типов не должна представлять сложностей, текущая ситуация такова: PEP 484 неявно рекомендует пользователям сторониться (*https://peps.python.org/pep-0484/#the-numeric-tower*) числовой башни в аннотациях типов, а средствам проверки типов зашивать в код отношения тип–подтип между встроенными типами `complex`, `float` и `int`. Муру так и поступает, а кроме того, прагматично принимает, что типы `int` и `float` *совместимы с* `SupportsComplex`, хотя и не реализуют метод `__complex__`.

Я наткнулся на неожиданности только при использовании `isinstance` для проверки числовых протоколов `Supports*` во время экспериментов с преобразованием в `complex` и обратно. Если вы не работаете с комплексными числами, то можете смело полагаться на эти протоколы вместо ABC из пакета `numbers`.

---

[1] Я не проверял все варианты целых и чисел с плавающей точкой, предоставляемые NumPy.

[2] Все числовые типы NumPy зарегистрированы как виртуальные подклассы соответствующих ABC из пакета `numbers`, которые Муру игнорирует.

[3] Это ложь во спасение со стороны typeshed: в версии Python 3.9 встроенный тип `complex` на самом деле не имеет метода `__complex__`.

Сформулируем основные уроки этого раздела.

- ABC из пакета `numbers` можно использовать для проверки типов во время выполнения, но для статической типизации они не годятся.
- Статические числовые протоколы `SupportsComplex`, `SupportsFloat` и т. д. хорошо работают со статической типизацией, но ненадежны при проверке во время выполнения типов, связанных с комплексными числами.

Теперь мы готовы подвести краткие итоги этой главы.

Резюме

Карта типизации (рис. 13.1) – ключ к этой главе. После краткого введения во все четыре подхода к типизации мы сравнили динамические и статические протоколы, поддерживающие соответственно утиную и статическую утиную типизации. У обоих видов протоколов есть общая черта: от класса никогда не требуется явно объявлять поддержку конкретного протокола. Для поддержки протокола достаточно, если класс реализует необходимые методы.

Следующим важным разделом было «Программирование уток», где мы выяснили, до каких пределов готов дойти интерпретатор Python в стремлении обеспечить работоспособность динамических протоколов последовательностей и итерируемых объектов, пусть даже частичную. Далее мы видели, как заставить класс реализовывать протокол во время выполнения, добавив дополнительные методы посредством партизанского латания. Мы завершили раздел об утиной типизации советами на тему защитного программирования, в частности о том, как обнаруживать структурные типы, не прибегая к явным проверкам с помощью функций `isinstance` или `hasattr`, а ограничиваясь конструкцией `try/except` и принципом быстрого отказа.

Познакомившись с гусиной типизацией в эссе Алекса Мартелли «Водоплавающие птицы и ABC», мы увидели, как порождать подклассы существующих ABC, обсудили важные ABC в стандартной библиотеке и создали с нуля свой ABC, который затем реализовали двумя способами: с помощью традиционных подклассов и регистрации виртуальных подклассов. В завершение этого раздела мы показали, как специальный метод `__subclasshook__` позволяет ABC поддержать структурную типизацию путем распознавания не связанных между собой классов, которые предоставляют методы, требуемые интерфейсом, определенным в ABC.

Последним крупным разделом был «Статические протоколы», где мы вернулись к рассмотрению статической утиной типизации, начатому в главе 8. Мы видели, что декоратор `@runtime_checkable` также пользуется методом `__subclasshook__` для поддержки структурной типизации во время выполнения – хотя использовать статические протоколы все же рекомендуется в сочетании со средствами статической проверки типов, которые принимают во внимание аннотации типов, чтобы повысить надежность структурной типизации. Далее мы поговорили о проектировании и кодировании статического протокола и о том, как его расширять. В заключительном разделе «ABC из пакета numbers и числовые протоколы» мы рассказали печальную историю о рухнувшей числовой башне и о нескольких недостатках предлагаемой альтернативы: статических числовых протоколах типа `SupportsFloat` и других, добавленных в модуль `typing` в версии Python 3.8.

Главный посыл этой главы – то, что в современном Python есть четыре взаимодополняющих способа программирования с помощью интерфейсов, каждый со своими достоинствами и недостатками. Вы без сомнения найдете примеры использования всех схем типизации в любой современной кодовой базе на Python сколько-нибудь значительного размера. Не стоит отвергать ни один из этих подходов, поскольку тем самым вы без нужды осложните себе работу.

При всем при том Python завоевал широкую популярность, когда поддерживал только утиную типизацию. Другие популярные языки, например JavaScript, PHP и Ruby, а также Lisp, Smalltalk, Erlang и Clojure – не такие популярные, но очень влиятельные – раньше и теперь утверждают свое влияние, задействуя всю мощь и простоту утиной типизации.

Дополнительная литература

В качестве краткого обзора плюсов и минусов типизации, а также важности `typing.Protocol` для поддержания статически проверяемых кодовых баз в работоспособном состоянии я настоятельно рекомендую статью Глифа Лефковица «I Want A New Duck: typing.Protocol and the future of duck typing» (*https://glyph.twistedmatrix.com/2020/07/new-duck.html*). Я также многое узнал из его статьи «Interfaces and Protocols» (*https://glyph.twistedmatrix.com/2021/03/interfaces-and-protocols.html*), в которой `typing.Protocol` сравнивается с `zope.interface`, прежним механизмом определения интерфейсов в слабо связанных системах с плагинами, который используется в CMS Plone (*https://plone.org/*), веб-каркасе Pyramid (*https://trypyramid.com/*) и каркасе асинхронного программирования Twisted (*https://twistedmatrix.com/trac/*) – проекте, который запустил Глиф[1].

Во всех хороших книгах о Python – почти по определению – много внимания уделено утиной типизации. Из моих любимых книг две были переизданы после выхода первого издания этой книги: Naomi Ceder «The Quick Python Book», 3-е издание (Manning), и Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», 3-е издание (O'Reilly).

Аргументы за и против динамической типизации прозвучали в интервью, данном Гвидо ван Россумом Биллу Веннерсу и опубликованном на странице «Контракты в Python: беседа с Гвидо ван Россумом, часть IV» (*http://www.artima.com/intv/pycontract.html*). Познавательный и сбалансированный вклад в этот спор – статья Мартина Фаулера «Dynamic Typing» (*https://martinfowler.com/bliki/DynamicTyping.html*). Он же написал статью «Role Interface» (*https://martinfowler.com/bliki/RoleInterface.html*), упомянутую мной в разделе «Рекомендации по проектированию протоколов». Хотя она и не об утиной типизации, но имеет самое непосредственное отношение к проектированию протоколов в Python, поскольку автор сравнивает узкие ролевые интерфейсы с более широкими открытыми интерфейсами классов вообще.

Документация по Муру зачастую является лучшим источником информации обо всем связанном со статической типизацией в Python, включая статическую утиную типизацию, которая рассматривается в главе «Protocols and structural subtyping» (*https://mypy.readthedocs.io/en/stable/protocols.html*).

---

[1] Благодарю технического рецензента Юрген Гмаха, порекомендовавшего мне статью «Interfaces and Protocols».

Все остальные ссылки относятся к гусиной типизации. В книге Beazley, Jones «*Python Cookbook*», 3-е издание (O'Reilly), есть раздел, посвященный определению ABC (рецепт 8.12). Эта книга была написана до выхода версии Python 3.4, поэтому в ней используется синтаксис с именованным параметром `metaclass`, а не рекомендуемое сейчас объявление ABC с помощью наследования `abc.ABC`. Но если не считать эту мелкую деталь, в рецепте прекрасно описаны все основные особенности ABC, а завершается он советом, процитированным в конце предыдущего раздела.

В книге Дуга Хеллманна «The Python Standard Library by Example» (Addison-Wesley) есть глава о модуле `abc`. Она опубликована также на великолепном сайте Дуга PyMOTW – Python Module of the Week (*http://pymotw.com/2/abc/index.html*). Хеллманн пользуется старым стилем объявления ABC: `PluginBase(metaclass=abc.ABCMeta)` вместо более простого `PluginBase(abc.ABC)`, появившегося в версии Python 3.4.

При работе с ABC множественное наследование не только часто встречается, но и практически неизбежно, поскольку все фундаментальные ABC коллекций – `Sequence`, `Mapping` и `Set` – расширяют несколько ABC (см. рис. 13.4). Поэтому глава 14 станет важным дополнением к этой.

В документе PEP 3119 «Introducing Abstract Base Classes» (*https://www.python.org/dev/peps/pep-3119*) приводится обоснование ABC, а в документе PEP 3141 «A Type Hierarchy for Numbers» (*https://www.python.org/dev/peps/pep-3141*) описываются ABC из модуля `numbers` (*https://docs.python.org/3/library/numbers.html*), но обсуждение в проблеме Муру #3186 «int is not a Number?» (*https://github.com/python/mypy/issues/3186*) содержит дополнительные аргументы по поводу того, почему числовая башня не годится для статической проверки типов. Алекс Уэйгуд написал развернутый ответ на сайте StackOverflow (*https://stackoverflow.com/questions/69334475/how-to-hint-at-number-types-i-e-subclasses-of-number-not-numbers-themselv/69383462#69383462*), в котором обсуждаются способы аннотирования числовых типов. Я буду поглядывать, что пишут в обсуждении проблемы Муру #3186 в надежде на счастливый конец, который сделает статическую и гусиную типизации совместимыми – какими они и должны быть.

Поговорим

Путешествие в мир статической типизации в Python на машине MVP

Я работаю в компании Thoughtworks, мировом лидере в области разработки гибкого (agile) программного обеспечения. Мы часто рекомендуем своим клиентам создавать и развертывать MVP (minimal viable product): минимально жизнеспособный продукт: «простую версию продукта, передаваемую пользователям с целью подтвердить основные бизнес-предположения», согласно определению моего коллеги Пауло Кароли в статье «Lean Inception» (*https://martinfowler.com/articles/lean-inception/*), опубликованной в коллективном блоге Мартина Фаулера.

Гвидо ван Россум и другие разработчики ядра, которые спроектировали и реализовали статическую типизацию, следовали стратегии MVP начиная с 2006 года. Сначала в версии Python 3.0 был реализован документ PEP 3107 «Function Annotations» (*https://peps.python.org/pep-3107/*) с очень ограниченной

семантикой: только синтаксис, необходимый для присоединения аннотаций к параметрам и возвращаемым значениям функций. При этом явно ставилась цель – провести эксперимент и собрать отзывы, а это и есть ключевые достоинства MVP.

Спустя восемь лет был предложен и одобрен документ PEP 484 «Type Hints» (*https://peps.python.org/pep-0484/*). Его реализация в Python 3.5 не потребовала никаких изменений в языке или в стандартной библиотеке, кроме добавления модуля `typing`, от которого никакая часть стандартной библиотеки не зависит. PEP 484 поддерживал только номинальные типы в обобщенных типах – по аналогии с Java, – но фактически статическая проверка производилась внешними инструментами. Отсутствовали такие важные возможности, как аннотации переменных, встроенные обобщенные типы и протоколы. Несмотря на ограничения, этот MVP оказался достаточно ценным для привлечения инвестиций и внедрения компаниями с очень большими базами кода на Python, в т. ч. Dropbox, Google и Facebook. Его поддержка была включена и в профессиональные IDE, например PyCharm (*https://www.jetbrains.com/pycharm/*), Wing (*https://wingware.com/*) и VS Code (*https://code.visualstudio.com/*).

Документ PEP 526 «Syntax for Variable Annotations» (*https://peps.python.org/pep-0526/*) стал первым шагом эволюции, потребовавшим внесения изменений в интерпретатор Python 3.6. Дополнительные изменения были внесены в интерпретатор Python 3.7 ради поддержки документов PEP 563 «Postponed Evaluation of Annotations» (*https://peps.python.org/pep-0563/*) и PEP 560 «Core support for typing module and generic types» (*https://peps.python.org/pep-0560/*), которые позволяли встроенным и библиотечным коллекциям принимать обобщенные типы в аннотациях в Python 3.9 – благодаря документу PEP 585 «Type Hinting Generics In Standard Collections» (*https://peps.python.org/pep-0585/*).

За эти годы некоторых пользователей Python, включая меня, поддержка типизации разочаровала. После изучения Go отсутствие статической утиной типизации в Python казалось мне непонятным, ведь это язык, в котором утиная типизация всегда была главным достоинством.

Но такова природа MVP: возможно, они не удовлетворяют всех потенциальных пользователей, но зато могут быть реализованы ценой меньших усилий, а после получения отзывов по результатам практического применения дальнейшая разработка пойдет в правильном направлении.

Если нужно назвать всего один урок, который мы вынесли из Python 3, то это тот факт, что постепенный прогресс безопаснее выпуска новых версий, включающих все-все-все. Я рад, что нам не пришлось ждать выхода Python 4, – если таковой когда-нибудь состоится, – чтобы сделать Python более привлекательным для крупных корпораций, считающих, что преимущества статической типизации перевешивают дополнительную сложность.

Подходы к типизации в популярных языках

На рис. 13.8 приведен вариант карты типизации (рис. 13.1) с названиями некоторых популярных языков, поддерживающих каждый из подходов к типизации.

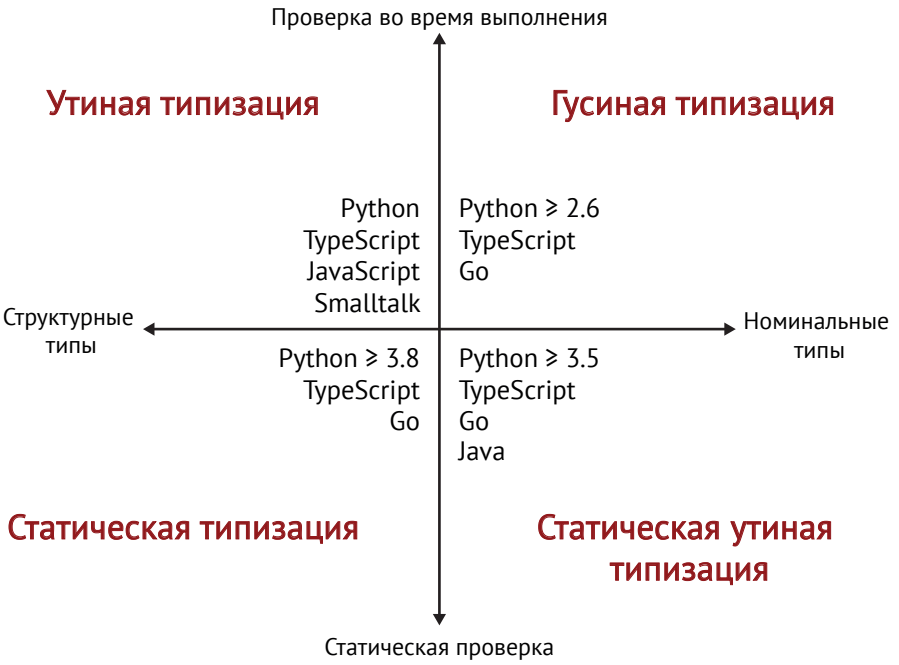


Рис. 13.8. Четыре подхода к проверке типов и некоторые языки, в которых они поддерживаются

TypeScript и Python ⩾ 3.8 – единственные языки в моей небольшой и произвольной выборке, которые поддерживают все четыре подхода.

Go – очевидно, статически типизированный язык в традиции Pascal, но в нем впервые была применена статическая утиная типизация – по крайней мере, среди языков, широко распространенных в настоящее время. Я также поместил Go в квадрант гусиной типизации из-за имеющихся в нем утверждений относительно типов, которые открывают возможность для проверки и адаптации к различным типам во время выполнения.

Если бы я рисовал подобную диаграмму в 2000 году, то какие-то языки присутствовали бы только в квадрантах утиной и статической типизаций. Мне неизвестны языки, поддерживавшие статическую утиную или гусиную типизации 20 лет назад. Тот факт, что в каждом из четырех квадрантов есть как минимум три популярных языка, позволяет предположить, что каждый из четырех подходов к типизации находит немало приверженцев.

Партизанское латание

У партизанского латания плохая репутация. Если им злоупотреблять, то можно получить систему, трудную для понимания и сопровождения. Заплата обычно тесно связана с конечным объектом, что делает его хрупким. Другая проблема состоит в том, что в случае партизанского латания двух библиотек возможны конфликты, в результате которых библиотека, загруженная второй, затрет заплаты, внесенные первой.

Но партизанское латание может также принести пользу, например чтобы добавить в класс реализацию протокола на этапе выполнения. Паттерн проектирования Адаптер решает ту же проблему путем реализации нового класса.

Код на Python легко поддается партизанскому латанию с некоторыми ограничениями. В отличие от Ruby и JavaScript, Python не позволяет латать встроенные типы. Лично я считаю это плюсом, так как есть уверенность, что объект `str` всегда будет иметь один и тот же набор методов. Это ограничение снижает шансы на то, что внешние библиотеки попытаются применить конфликтующие заплаты.

Метафоры и идиомы в интерфейсах

Метафора способствует пониманию, делая ясными возможности и ограничения. В этом ценность слов «стек»[1] и «очередь» – в описании соответствующих фундаментальных структур данных: благодаря им понятно, как добавляются и удаляются элементы. С другой стороны, Алан Купер (Alan Cooper) в книге «About Face, the Essentials of Interaction Design»[2], издание 4 (Wiley), пишет:

> Строгая приверженность метафорам слишком тесно – без всякой на то необходимости – связывает интерфейсы с явлениями материального мира.

Он имел в виду пользовательские интерфейсы, однако совет в равной мере относится и к API. Но Купер благосклонно относится к «действительно подходящим» метафорам, «будто упавшим с небес», и не возражает против их использования (он пишет «будто упавшим с небес», потому что найти хорошую метафору настолько трудно, что не стоит тратить времени на их целенаправленный поиск). Мне кажется, что образ машины для игры в бинго, который я использовал в этой главе, удачен, и я остался верен ему.

«About Face» – пожалуй, лучшая из прочитанных мной книг о пользовательском интерфейсе, – а я прочел не только ее. И одна из самых ценных мыслей, почерпнутых мной у Купера, – отказ от использования метафор как парадигмы дизайна и замена их «идиоматическими интерфейсами».

В «About Face» Купер говорит не об API, но чем дольше я размышляю о его идеях, тем больше мне кажется, что они применимы и к Python. Фундаментальные протоколы языка – это то, что Купер называет «идиомами». Однажды поняв, что такое «последовательность», мы можем применять это знание в разных контекстах. Это и есть главная тема моей книги: выявление фундаментальных идиом языка, что позволяет сделать код кратким, эффективным и удобочитаемым – для мастера-питониста.

---

[1] Слово «stack» (букв. стопка) на заре развития программирования в СССР переводили как «магазин» (термин до сих пор сохранился в теории автоматов), и это была очевидная метафора автоматного рожка. Жаль, что теперь его переводят бесцветной калькой «стек». – *Прим. перев.*

[2] *Алан Купер.* Интерфейс, основы проектирования взаимодействия. 4-е изд. Питер, 2022.

Глава 14

Наследование: к добру или к худу

> [...] нам нужна была (и до сих пор нужна) более качественная теория наследования вообще. Например, наследование и инстанцирование (instancing) (тоже разновидность наследования) мешают в одну кучу прагматику (например, выделение общего кода для экономии памяти) и семантику (используемую для слишком многих задач, как то: специализация, обобщение, видообразование и т. д.).
>
> – Алан Кэй, «The Early History of Smalltalk»[1]

Эта глава посвящена наследованию и подклассам. Я предполагаю, что читатель знаком с этими идеями, например из чтения «Пособия по языку Python» (*https://docs.python.org/3/tutorial/classes.html*) или по опыту работы с каким-нибудь другим популярным объектно-ориентированным языком, например Java, C# или C++. В центре нашего внимания будут четыре специфические особенности Python:

- функция `super()`;
- проблемы наследования встроенным типам;
- множественное наследование и порядок разрешения методов;
- классы-примеси.

Множественное наследование – это способность класса иметь более одного базового класса. C++ его поддерживает, а Java и C# нет. Многие считают, что множественное наследование порождает больше проблем, чем решает. Оно было сознательно изъято из Java после печального опыта злоупотребления в ранних кодовых базах на C++.

В этой главе введение в множественное наследование написано для тех, кто никогда им раньше не пользовался. Приводятся рекомендации, как работать с одиночным или множественным наследованием, если без него не обойтись.

По состоянию на 2021 год имеет место отрицательное отношение к чрезмерному использованию наследования вообще – не только множественного,

---

[1] *Alan Kay.* The Early History of Smalltalk. Опубликовано в SIGPLAN Not. 28, 3 (март 1993), 69–95. Имеется также в сети (*http://propella.sakura.ne.jp/earlyHistoryST/EarlyHistoryST.html*). Спасибо моему другу Кристиано Андерсону, который прислал эту ссылку, когда я работал над данной главой.

потому что подклассы и суперклассы тесно связаны. Тесная связанность означает, что изменения в одной части программы могут иметь неожиданные и далеко идущие последствия в других ее частях, что делает систему хрупкой и трудной для понимания.

Однако нам так или иначе приходится сопровождать уже написанные системы, основанные на сложных иерархиях классов, или пользоваться каркасами, заставляющими прибегать к наследованию, иногда даже множественному.

Мой рассказ о практическом применении множественного наследования иллюстрируется примерами из стандартной библиотеки, веб-каркаса Django и библиотеки пользовательского интерфейса Tkinter.

Что нового в этой главе

В Python не появилось никаких новых средств, относящихся к теме этой главы, но я значительно отредактировал ее, прислушавшись к мнениям технических рецензентов второго издания, особенно Леонардо Рохаэля и Калеба Хэттинга.

Я написал новый вводный раздел, посвященный встроенной функции `super()`, и изменил примеры в разделе «Множественное наследование и порядок разрешения методов», чтобы глубже исследовать, как `super()` поддерживает *кооперативное множественное наследование*.

Раздел «Классы-примеси» также написан заново. Раздел «Множественное наследование в реальном мире» реорганизован, в нем мы рассматриваем простые примеры примесей из стандартной библиотеки, прежде чем переходить к сложным иерархиям Django и Tkinter.

Как следует из названия, подводные камни наследования всегда были одной из основных тем данной главы. Но все больше разработчиков начинают считать наследование настолько проблематичным, что я добавил пару абзацев о том, как избегать его, в конце разделов «Резюме» и «Дополнительная литература».

Начнем с краткого рассказа о таинственной функции `super()`.

Функция super()

Единообразное использование встроенной функции `super()` очень важно для удобства сопровождения объектно-ориентированных программ на Python.

Метод суперкласса, переопределенный в подклассе, обычно должен вызывать соответствующий метод суперкласса. Ниже показано, как рекомендуется это делать; пример взят из документации по модулю *collections*, раздел «Пример и рецепты использования OrderedDict» (*https://docs.python.org/3/library/collections.html# ordereddict-examples-and-recipes*)[1]:

```
class LastUpdatedOrderedDict(OrderedDict):
    """Элементы хранятся в порядке, определяемом последним обновлением"""
    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

[1] Я изменил в примере только строку документации, потому что оригинальная вводила в заблуждение. В ней было написано: «Элементы хранятся в порядке добавления ключей», но это не то, о чем недвусмысленно говорит имя класса `LastUpdatedOrderedDict`.

Для выполнения своей работы класс `LastUpdatedOrderedDict` переопределяет метод `__setitem__` следующим образом:

1. Воспользоваться конструкцией `super().__setitem__`, чтобы вызвать метод суперкласса, который вставит или обновит пару ключ-значение.
2. Вызвать метод `self.move_to_end`, чтобы переместить измененную запись с ключом `key` в конец словаря.

Вызов переопределенного метода `__init__` особенно важен, поскольку позволяет суперклассам принять участие в инициализации экземпляра.

Если вы изучали объектно-ориентированное программирование на Java, то, возможно, помните, что в Java конструктор автоматически вызывает конструктор без аргументов суперкласса. Python этого не делает. Вы должны привыкнуть к такой шаблонной последовательности:

```
def __init__(self, a, b) :
    super().__init__(a, b)
    ... # дополнительная инициализация
```

Возможно, вам попадался код, в котором не используется `super()`, а вместо этого непосредственно вызывается метод суперкласса:

```
class NotRecommended(OrderedDict):
    """Так поступать не рекомендуется!"""

    def __setitem__(self, key, value):
        OrderedDict.__setitem__(self, key, value)
        self.move_to_end(key)
```

В данном случае этот код работает, но такая практика не рекомендуется по двум причинам. Во-первых, в коде зашито имя базового класса. Имя `OrderedDict` встречается в предложении `class` и внутри `__setitem__`. Тот, кто в будущем изменит базовый класс или добавит еще один, может забыть о том, что необходимо изменить и тело `__setitem__` – вот вам и ошибка.

Вторая причина в том, что `super` реализует логику обработки иерархий с множественным наследованием. Мы вернемся к этой теме в разделе «Множественное наследование и порядок разрешения методов». И в заключение этого краткого обзора `super` полезно напомнить, как мы должны были вызывать ее в Python 2, поскольку старая сигнатура с двумя аргументами проливает свет на принцип работы этой функции:

```
class LastUpdatedOrderedDict(OrderedDict):
    """Этот код работает в Python 2 и в Python 3"""

    def __setitem__(self, key, value):
        super(LastUpdatedOrderedDict, self).__setitem__(key, value)
        self.move_to_end(key)
```

Теперь оба аргумента `super` факультативны. Компилятор байт-кода в Python 3 автоматически подставляет их, исследуя объемлющий контекст вызова `super()` в методе. Аргументы имеют следующую семантику:

`type`

Начало пути поиска суперкласса, реализующего нужный метод. По умолчанию это класс, которому принадлежит метод, откуда вызвана `super()`.

`object_or_type`

Объект (в случае вызова методов экземпляра) или класс (в случае вызова методов класса), от имени которого вызывается метод. По умолчанию это `self`, если вызов `super()` встречается в методе экземпляра.

Кто бы ни предоставил эти аргументы – вы или компилятор, – вызов `super()` возвращает динамический прокси-объект, который находит метод (в нашем примере `__setitem__`) в суперклассе, определяемом параметром `type`, и связывает его с `object_or_type`, так что нам не нужно явно передавать объект (`self`), от имени которого вызывается метод.

В Python 3 мы по-прежнему можем явно передать `super()` первый и второй аргументы[1]. Но нужны они только в специальных случаях, например чтобы пропустить часть MRO при тестировании или отладке или обойти нежелательное поведение в суперклассе.

Теперь обсудим, с какими подводными камнями можно столкнуться при наследовании встроенным типам.

Сложности наследования встроенным типам

В ранних версиях Python создать подкласс встроенного типа, например `list` или `dict`, было невозможно. Начиная с Python 2.2 такая возможность появилась, но с существенной оговоркой: код встроенного типа (написанный на C) обычно не вызывает специальные методы, переопределенные в пользовательских классах. Суть проблемы хорошо описана в документации по интерпретатору *PyPy*, в главе «Различия между PyPy и CPython», раздел «Подклассы встроенных типов» (*https://doc.pypy.org/en/latest/cpython_differences.html#subclasses-of-built-in-types*):

> Официально в CPython нет никаких правил, определяющих, когда переопределенный в подклассе метод встроенного типа вызывается и вызывается ли он вообще. В качестве приближения к истине можно считать, что такие методы никогда не вызываются другими встроенными методами того же объекта. Например, метод `__getitem__()`, переопределенный в подклассе `dict`, не будет вызываться из встроенного метода `get()`.

Проблема иллюстрируется в примере 14.1.

Пример 14.1. Наш метод `__setitem__` игнорируется методами `__init__` и `__update__` встроенного типа `dict`

```
>>> class DoppelDict(dict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2) ❶
...
>>> dd = DoppelDict(one=1) ❷
```

[1] Можно также задавать только первый аргумент, но это вредно и вскоре может быть объявлено нерекомендуемой практикой с благословения Гвидо ван Россума, который и является автором функции `super()`. См. обсуждение в статье «Is it time to deprecate unbound super methods?» (*https://discuss.python.org/t/is-it-time-to-deprecate-unbound-super-methods/1833*).

```
>>> dd
{'one': 1}
>>> dd['two'] = 2 ❸
>>> dd
{'one': 1, 'two': [2, 2]}
>>> dd.update(three=3) ❹
>>> dd
{'three': 3, 'one': 1, 'two': [2, 2]}
```

❶ Метод `DoppelDict.__setitem__` повторяет значение при сохранении (только для того, чтобы его эффект был наглядно виден). Свою работу он делегирует методу суперкласса.
❷ Метод `__init__`, унаследованный от `dict`, очевидно, не знает, что `__setitem__` переопределен: значение `'one'` не повторено.
❸ Оператор `[]` вызывает наш метод `__setitem__` и работает, как и ожидалось: `'two'` отображается на повторенное значение `[2, 2]`.
❹ Метод `update` класса `dict` также не пользуется нашей версией `__setitem__`: значение `'three'` не повторено.

Поведение встроенных типов находится в явном противоречии с основным правилом объектно-ориентированного программирования: поиск методов всегда должен начинаться с класса самого объекта (`self`), даже если он вызывается из метода, реализованного в суперклассе. Это явление называется «поздним связыванием», и Алан Кэй – из зала славы Smalltalk – считает его ключевым свойством объектно-ориентированного программирования: в любом вызове вида `x.method()` конкретный вызываемый метод должен определяться на этапе выполнения в зависимости от класса `x`[1]. Это печальное положение дел усугубляет проблемы, которые были описаны в разделе «Несогласованное использование __missing__ в стандартной библиотеке» главы 3.

Проблема не ограничивается вызовами изнутри объекта – когда `self.get()` вызывает `self.__getitem__()`, – но возникает и для переопределенных методов других классов, которые должны вызываться из встроенных методов. Пример 14.2 основан на примере из документации по PyPy (*https://doc.pypy.org/en/latest /cpython_differences.html#subclasses-of-built-in-types*).

Пример 14.2. Метод `__getitem__` из класса `AnswerDict` игнорируется методом `dict.update`

```
... def __getitem__(self, key): ❶
...     return 42
...
>>> ad = AnswerDict(a='foo') ❷
>>> ad['a'] ❸
42
>>> d = {}
>>> d.update(ad) ❹
>>> d['a'] ❺
```

[1] Интересно, что в C++ есть понятие виртуальных и невиртуальных методов. Для виртуальных методов применяется позднее связывание, а невиртуальные связываются на этапе компиляции. Хотя все методы, которые мы можем написать на Python, связываются поздно, как виртуальные методы, встроенные объекты, написанные на C, похоже, по умолчанию имеют невиртуальные методы, по крайней мере в CPython.

```
'foo'
>>> d
{'a': 'foo'}
```

❶ Метод `AnswerDict.__getitem__` для любого ключа возвращает 42.
❷ `ad` – экземпляр `AnswerDict`, инициализированный парой `('a', 'foo')`.
❸ `ad['a']` возвращает 42, как и ожидалось.
❹ `d` – экземпляр класса `dict`, обновленный объектом `ad`.
❺ Метод `dict.update` игнорирует наш метод `AnswerDict.__getitem__`.

Прямое наследование таким встроенным типам, как `dict`, `list` или `str`, чревато ошибками, потому что встроенные методы, как правило, игнорируют написанные пользователем переопределенные методы. Вместо создания подклассов встроенных объектов наследуйте свои классы от классов в модуле `collections` (*http://docs.python.org/3/library/collections.html*) – `UserDict`, `UserList` и `UserString`, которые специально предназначены для беспроблемного наследования.

Если наследовать подклассу `collections.UserDict`, а не `dict`, то проблемы, продемонстрированные в примерах 14.1 и 14.2, исчезают. См. пример 14.3.

Пример 14.3. Классы `DoppelDict2` и `AnswerDict2` работают, как и ожидалось, потому что расширяют `UserDict`, а не `dict`

```
>>> import collections
>>>
>>> class DoppelDict2(collections.UserDict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)
...
>>> dd = DoppelDict2(one=1)
>>> dd
{'one': [1, 1]}
>>> dd['two'] = 2
>>> dd
{'two': [2, 2], 'one': [1, 1]}
>>> dd.update(three=3)
>>> dd
{'two': [2, 2], 'three': [3, 3], 'one': [1, 1]}
>>>
>>> class AnswerDict2(collections.UserDict):
... def __getitem__(self, key):
... return 42
...
>>> ad = AnswerDict2(a='foo')
>>> ad['a']
42
>>> d = {}
>>> d.update(ad)
>>> d['a']
42
>>> d
{'a': 42}
```

Для оценки дополнительных усилий на создание подкласса встроенного типа я переписал класс `StrKeyDict` из примера 3.9, так чтобы он наследовал `dict`, а не `UserDict`. Чтобы проходили те же тесты, что и раньше, мне пришлось реализовать методы `__init__`, `get` и `update`, потому что их версии, унаследованные от `dict`, отказывались признавать переопределенные методы `__missing__`, `__contains__` и `__setitem__`. В подклассе `UserDict` из примера 3.8 было 16 строк, а в экспериментальном подклассе `dict` – целых 33 строки[1].

Подведем итоги: описанная в этом разделе проблема относится только к делегированию методов встроенных типов, написанных на языке C, и только к пользовательским подклассам этих типов. Если наследовать классу, написанному на Python, например `UserDict` или `MutableMapping`, то эта проблема не возникает[2].

Теперь перейдем к вопросу, возникающему в связи со множественным наследованием: если у класса есть два суперкласса, то как Python решает, какой атрибут использовать, когда вызывается `super().attr`, но в обоих суперклассах есть атрибуты с таким именем?

Множественное наследование и порядок разрешения методов

Любой язык с множественным наследованием должен как-то разрешать конфликты имен в случае, когда в не связанных между собой родительских классах имеются методы с одним и тем же именем. Эта «проблема ромбовидного наследования» иллюстрируется на рис. 14.1 и в примере 14.4.

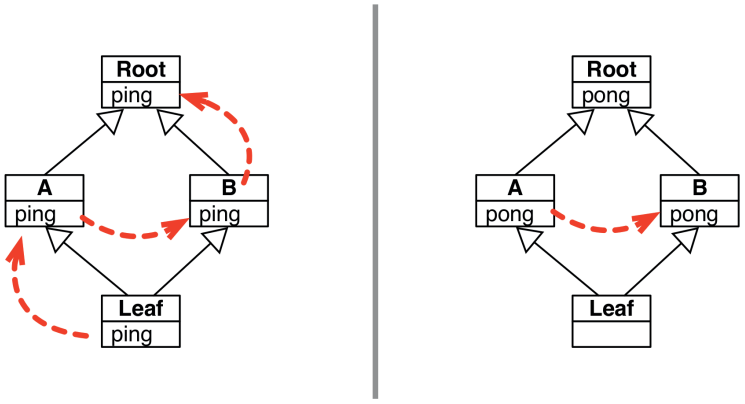


Рис. 14.1. Слева: последовательность активации для вызова `leaf1.ping()`. Справа: последовательность активации для вызова `leaf1.pong()`

[1] Для любознательных читателей – экспериментальная версия находится в файле *14-inheritance/strkeydict_dictsub.py* в репозитории кода к этой книге по адресу *https://github.com/fluentpython/example-code-2e*.

[2] Кстати говоря, в этом отношении PyPy ведет себя «корректнее», чем CPython, но ценой незначительной несовместимости. См. раздел «Различия между PyPy и CPython» (*https://doc.pypy.org/en/latest/cpython_differences.html#subclasses-of-built-in-types*).

Пример 14.4. diamond.py: классы `Leaf`, `A`, `B`, `Root` образуют граф, показанный на рис. 14.1

```
class Root: ❶
    def ping(self):
        print(f'{self}.ping() in Root')

    def pong(self):
        print(f'{self}.pong() in Root')

    def __repr__(self):
        cls_name = type(self).__name__
        return f'<instance of {cls_name}>'

class A(Root): ❷
    def ping(self):
        print(f'{self}.ping() in A')
        super().ping()

    def pong(self):
        print(f'{self}.pong() in A')
        super().pong()

class B(Root): ❸
    def ping(self):
        print(f'{self}.ping() in B')
        super().ping()

    def pong(self):
        vprint(f'{self}.pong() in B')

class Leaf(A, B): ❹
    def ping(self):
        print(f'{self}.ping() in Leaf')
        super().ping()
```

❶ `Root` предоставляет методы `ping`, `pong` и `__repr__`, чтобы вывод было проще читать.
❷ Оба метода `ping` и `pong` в классе A вызывают `super()`.
❸ В классе `B` только метод `ping` вызывает `super()`.
❹ Класс `Leaf` реализует только метод `ping`, который вызывает `super()`.

Теперь посмотрим, что происходит при вызове методов `ping` и `pong` экземпляра `Leaf` (пример 14.5).

Пример 14.5. Тесты вызова методов `ping` и `pong` объекта `Leaf`

```
>>> leaf1 = Leaf() ❶
>>> leaf1.ping() ❷
<instance of Leaf>.ping() in Leaf
<instance of Leaf>.ping() in A
<instance of Leaf>.ping() in B
<instance of Leaf>.ping() in Root

>>> leaf1.pong() ❸
<instance of Leaf>.pong() in A
<instance of Leaf>.pong() in B
```

❶ `leaf1` – экземпляр `Leaf`.
❷ Вызов `leaf1.ping()` активирует методы `ping` в `Leaf`, `A`, `B` и `Root`, потому что методы `ping` в первых трех классах вызывают `super().ping()`.
❸ Вызов `leaf1.pong()` активирует метод `pong` в `A` благодаря наследованию, а тот вызывает `super().pong()`, что приводит к активации `B.pong`.

Последовательности активации, показанные в примерах 14.5 и на рис. 14.1, определяются двумя факторами:

- порядок разрешения методов в классе `Leaf`;
- использование `super()` в каждом методе.

В каждом классе имеется атрибут `__mro__`, в котором хранится кортеж ссылок на суперклассы в порядке разрешения методов, от текущего класса к классу `object`[1]. Для класса `Leaf` атрибут `__mro__` имеет вид:

```
>>> Leaf.__mro__ # doctest:+NORMALIZE_WHITESPACE
    (<class 'diamond1.Leaf'>, <class 'diamond1.A'>, <class 'diamond1.B'>,
     <class 'diamond1.Root'>, <class 'object'>)
```

Глядя на рис. 14.1, можно подумать, что MRO описывает поиск в ширину (*https://en.wikipedia.org/wiki/Breadth-first_search*), но это лишь совпадение для конкретной иерархии классов. MRO вычисляется опубликованным алгоритмом под названием C3. Его использование в Python подробно описано в статье Мишеля Симионато «The Python 2.3 Method Resolution Order» (*https://www.python.org/download/releases/2.3/mro/*). Это трудный текст, но Симионато пишет: «если вы не злоупотребляете множественным наследованием и не работаете с особо сложными иерархиями, то понимать алгоритм C3 необязательно и можно спокойно не читать эту статью».

MRO определяет только порядок активации, а будет ли конкретный метод активирован в каждом классе, зависит от того, вызывает реализация функцию `super()` или нет.

Рассмотрим эксперимент с методом `pong`. В классе `Leaf` он не переопределен, поэтому вызов `leaf1.pong()` активирует реализацию в следующем классе из `Leaf.__mro__`: классе `A`. Метод `A.pong` вызывает `super().pong()`. Следующим в MRO идет класс `B`, поэтому активируется `B.pong`. Но этот метод не вызывает `super().pong()`, поэтому последовательность активации здесь и оканчивается.

Порядок MRO принимает в расчет не только граф наследования, но также порядок, в котором перечислены суперклассы в объявлении подкласса. Иными словами, если бы в файле *diamond.py* (пример 14.4) класс `Leaf` был объявлен как `Leaf(B, A)`, то класс `B` встретился бы в `Leaf.__mro__` раньше `A`. Это повлияло бы на порядок активации методов `ping`, а кроме того, вызов `leaf1.pong()` активировал бы `B.pong` в силу наследования, но `A.pong` и `Root.pong` не получили бы управления, потому что `B.pong` не вызывает `super()`.

[1] В классах есть также метод `a.mro()`, но это продвинутая возможность, предназначенная для программирования метаклассов, которую мы кратко обсудим в разделе «Классы как объекты» главы 24. При нормальном использовании класса важно только содержимое атрибута `__mro__`.

Метод, вызывающий `super()`, называется *кооперативным*. Кооперативные методы позволяют организовать *кооперативное множественное наследование*. Эти термины выбраны не случайно: для работы множественного наследования в Python необходима активная кооперация участвующих методов. В классе `B` метод `ping` кооперативный, а метод `pong` нет.

Некооперативные методы могут стать причиной тонких ошибок. Многие программисты, прочитавшие код примера 14.4, возможно, ожидают, что когда метод `A.pong` вызывает `super.pong()`, тот в конечном итоге активирует `Root.pong`. Но если `B.pong` активирован раньше, то он «роняет мяч». Именно поэтому рекомендуется, чтобы каждый метод `m` некорневого класса вызывал `super().m()`.

Кооперативные методы должны иметь совместимые сигнатуры, потому что заранее неизвестно, будет ли `A.ping` вызван раньше или позже, чем `B.ping`. Последовательность активации зависит от порядка объявления классов `A` и `B` в каждом подклассе, наследующем им обоим.

Python – динамический язык, поэтому взаимодействие `super()` с MRO тоже динамическое. В примере 14.6 показан удивительный результат такого динамического поведения.

Пример 14.6. diamond2.py: классы, демонстрирующие динамическую природу `super()`

```
from diamond import A ❶

class U(): ❷
    def ping(self):
        print(f'{self}.ping() in U')
        super().ping() ❸

class LeafUA(U, A): ❹
    def ping(self):
        print(f'{self}.ping() in LeafUA')
        super().ping()
```

❶ Класс `A` берется из *diamond.py* (пример 14.4).
❷ Класс `U` не связан ни с `A`, ни с `Root` из модуля `diamond`.
❸ Что делает `super().ping()`? Ответ: зависит от ситуации. Читайте дальше.
❹ `LeafUA` наследует `U` и `A` именно в таком порядке.

Создав экземпляр `U` и попытавшись вызвать его метод `ping`, мы получим ошибку:

```
>>> u = U()
>>> u.ping()
Traceback (most recent call last):
   ...
AttributeError: 'super' object has no attribute 'ping'
```

У объекта `'super'`, возвращенного функцией `super()`, нет атрибута `'ping'`, потому что в MRO `U` есть два класса: `U` и `object`, а у последнего атрибут `'ping'` отсутствует.

Однако с методом `U.ping` не все так безнадежно. Попробуем:

```
>>> leaf2 = LeafUA()
>>> leaf2.ping()
<instance of LeafUA>.ping() in LeafUA
<instance of LeafUA>.ping() in U
<instance of LeafUA>.ping() in A
<instance of LeafUA>.ping() in Root
>>> LeafUA.__mro__ # doctest:+NORMALIZE_WHITESPACE
(<class 'diamond2.LeafUA'>, <class 'diamond2.U'>,
 <class 'diamond.A'>, <class 'diamond.Root'>, <class 'object'>)
```

Вызов `super().ping()` в `LeafUA` активирует `U.ping`, кооперативный метод, который вызывает `super().ping()`, что активирует `A.ping` и в конечном счете `Root.ping`.

Заметим, что базовыми классами `LeafUA` являются `(U, A)` в таком порядке. Если бы базовыми классами были `(A, U)`, то вызов `leaf2.ping()` так и не дошел бы до `U.ping`, потому что `super().ping()` в `A.ping` активировал бы `Root.ping`, а этот метод не вызывает `super()`.

В реальной программе класс, подобный `U`, был бы *классом-примесью*, т. е. классом, который предназначен для работы с другими классами при множественном наследовании с целью придать им дополнительную функциональность. Мы будем обсуждать эту тему в разделе «Классы-примеси».

Чтобы подвести итоги обсуждению MRO, я на рис. 14.2 изобразил часть сложного графа множественного наследования пакета Tkinter для построения пользовательских интерфейсов из стандартной библиотеки Python.

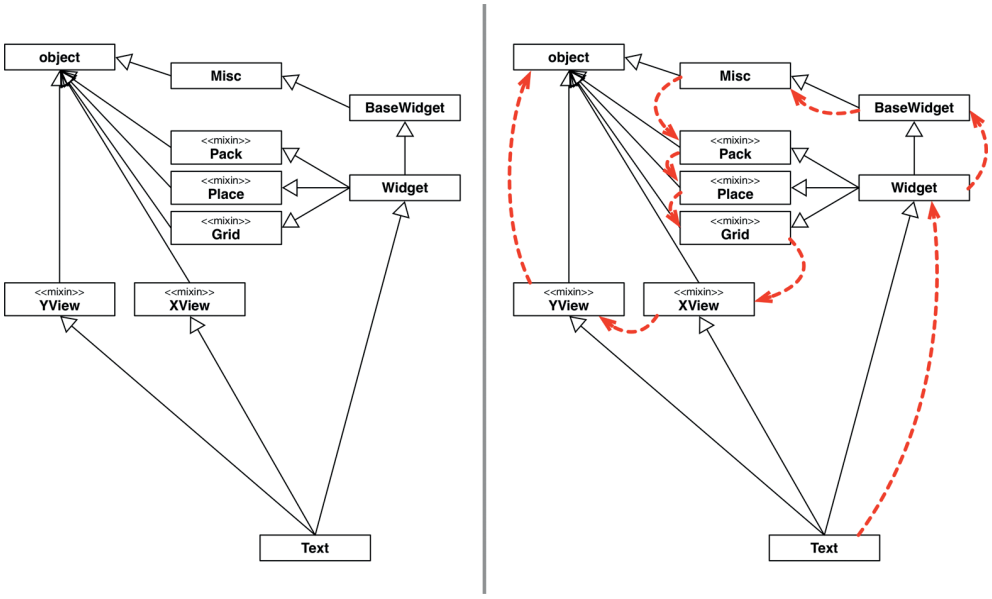


Рис. 14.2. Слева: UML-диаграмма класса виджета `Text` из пакета Tkinter и его суперклассов. Справа: пунктирными стрелками обозначен длинный и извилистый путь `Text.__mro__`

Изучая этот рисунок, начните с класса `Text` внизу. Класс `Text` реализует многострочный редактируемый текстовый виджет. Он обладает богатой функциональностью сам по себе, но еще и наследует многочисленные методы от дру-

гих классов. В левой части рисунка показана обычная UML-диаграмма классов. А справа она дополнена стрелками, показывающими порядок MRO, полученный с помощью вспомогательной функции `print_mro` из примера 14.7:

Пример 14.7. MRO класса `tkinter.Text`

```
>>> def print_mro(cls):
...     print(', '.join(c.__name__ for c in cls.__mro__))
>>> import tkinter
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

А теперь поговорим о примесях.

Классы-примеси

Класс-примесь предназначен для того, чтобы ему наследовали вместе с еще хотя бы одним классом при организации множественного наследования. Примесь не должна быть единственным базовым классом конкретного класса, потому что не предоставляет полную функциональность конкретного объекта, а лишь добавляет или настраивает поведение классов-потомков или братьев.

Классы-примеси – соглашение, не имеющее явной поддержки в Python и C++. Но Ruby позволяет явно определить и использовать модули, работающие как примеси, т. е. содержащие набор методов, которые можно включить для расширения функциональности класса. В C#, PHP и Rust реализованы классы-характеристики (traits), также представляющие собой явную форму примесей.

Рассмотрим простой, но практически полезный пример класса-примеси.

Отображения, не зависящие от регистра

В примере 14.8 показан класс `UpperCaseMixin`, который дает не зависящий от регистра доступ к отображениям с ключами-строками. Для этого он преобразует в верхний регистр ключи при добавлении и поиске.

Пример 14.8. uppermixin.py: `UpperCaseMixin` поддерживает отображения, не зависящие от регистра

```
import collections

def _upper(key): ❶
    try:
        return key.upper()
    except AttributeError:
        return key

class UpperCaseMixin: ❷
    def __setitem__(self, key, item):
        super().__setitem__(_upper(key), item)

    def __getitem__(self, key):
        return super().__getitem__(_upper(key))
```

```
    def get(self, key, default=None):
        return super().get(_upper(key), default)

    def __contains__(self, key):
        return super().__contains__(upper(key))
```

❶ Эта вспомогательная функция принимает ключ `key` любого типа и пытается вернуть `key.upper()`; если это не получается, то возвращает исходный ключ.
❷ Примесь реализует четыре основных метода отображений и всегда вызывает `super()` с ключом, преобразованным в верхний регистр (если это возможно).

Поскольку все методы `UpperCaseMixin` вызывают `super()`, эта примесь зависит от класса на том же уровне иерархии, который реализует или наследует методы с такой же сигнатурой. Чтобы внести свой вклад, примесь обычно должна находиться раньше других классов в MRO использующего ее подкласса. На практике это означает, что примеси должны располагаться в начале кортежа базовых классов в объявлении класса. В примере 14.9 показаны две ситуации.

Пример 14.9. uppermixin.py: два класса, использующих `UpperCaseMixin`

```
class UpperDict(UpperCaseMixin, collections.UserDict): ❶
    pass

class UpperCounter(UpperCaseMixin, collections.Counter): ❷
    """Специализированный 'Counter', который переводит строковые ключи в
    верхний регистр""" ❸
```

❶ `UpperDict` не нуждается в собственной реализации, но `UpperCaseMixin` должен быть первым базовым классом, иначе вызывались бы методы из `UserDict`.
❷ `UpperCaseMixin` работает также с `Counter`.
❸ Вместо `pass` лучше включить строку документации, чтобы удовлетворить синтаксическим требованиям к телу предложения `class`.

Ниже приведено несколько тестов для `UpperDict` из файла *uppermixin.py*:

```
>>> d = UpperDict([('a', 'letter A'), (2, 'digit two')])
>>> list(d.keys())
['A', 2]
>>> d['b'] = 'letter B'
>>> 'b' in d
True
>>> d['a'], d.get('B')
('letter A', 'letter B')
>>> list(d.keys())
['A', 2, 'B']
```

И демонстрация работы `UpperCounter`:

```
>>> c = UpperCounter('BaNanA')
>>> c.most_common()
[('A', 3), ('N', 2), ('B', 1)]
```

Классы `UpperDict` и `UpperCounter` кажутся почти магическими, но мне пришлось внимательно изучить код `UserDict` и `Counter`, чтобы заставить `UpperCaseMixin` действовать с ними заодно.

Например, в первой моей версии `UpperCaseMixin` не было метода `get`. Она работала с `UserDict`, но не с `Counter`. Класс `UserDict` наследует `get` от `collections.abc.Mapping`, и этот `get` вызывает `__getitem__`, реализованный мной. Но ключи не переводились в верхний регистр, когда `UpperCounter` загружался на стадии `__init__`. Это было связано с тем, что `Counter.__init__` вызывает `Counter.update`, который, в свою очередь, полагается на метод `get`, унаследованный от `dict`. Однако метод `get` в классе `dict` не вызывает `__getitem__`. Это и есть корень проблемы, обсуждавшейся в разделе «Несогласованное использование __missing__ в стандартной библиотеке» главы 3. Это также может служить яркой иллюстрацией хрупкого и подчас необъяснимого поведения программ, в которых используется наследование, пусть даже и в скромных масштабах.

В следующем разделе мы рассмотрим несколько примеров множественного наследования, зачастую с использованием классов-примесей.

Множественное наследование в реальном мире

В книге «Паттерны проектирования»[1] почти весь код написан на C++, но единственный пример множественного наследования дает паттерн Адаптер. В Python множественное наследование тоже не является нормой, но имеются важные примеры, которые я и хочу прокомментировать в этом разделе.

ABC – тоже примеси

В стандартной библиотеке Python множественное наследование особенно хорошо заметно в пакете `collections.abc`. И тут нет никакого противоречия: в конце концов, даже в Java поддерживается множественное наследование интерфейсов, а ABC – это объявление интерфейса, которое может содержать реализации конкретных методов[2].

В официальной документации по модулю `collections.abc` (*https://docs.python.org/3/library/collections.abc.html*) термин *подмешанный метод* (mixin method) употребляется для конкретных методов, реализованных во многих ABC коллекций. ABC, предоставляющие подмешанные методы, играют две роли: это определения интерфейсов и одновременно классы-примеси. Например, реализация `collections.UserDict` (*https://github.com/python/cpython/blob/8ece98a7e418c3c68a4c61bc47a2d0931b59a889/Lib/collections/__init__.py#L1084*) опирается на несколько подмешанных методов, предоставляемых классом `collections.abc.MutableMapping`.

ThreadingMixIn и ForkingMixIn

Пакет *http.server* (*https://github.com/python/cpython/blob/8ece98a7e418c3c68a4c61bc47a2d0931b59a889/Lib/collections/__init__.py#L1084*) предоставляет классы `HTTPServer`

---

[1] *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.* Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley).

[2] Выше уже упоминалось, что в Java 8 интерфейсам тоже разрешено предоставлять реализации методов. Эта новая возможность в официальном «Учебнике Java» называется «методы по умолчанию» (*https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html*).

и `ThreadingHTTPServer`. Последний был добавлен в версии Python 3.7. В его документации сказано:

class `http.server.ThreadingHTTPServer(server_address, RequestHandlerClass)`

> Этот класс идентичен `HTTPServer`, но для обработки запросов применяет потоки, используя `ThreadingMixIn`. Это полезно при работе с браузерами, которые заранее открывают сокеты, поскольку `HTTPServer` в этом случае ждал бы неопределенно долго.

Ниже приведен полный исходный код (*https://github.com/python/cpython/blob/ 17c23167942498296f0bdfffe52e72d53d66d693/Lib/http/server.py#L144*) класса `ThreadingHTTPServer` в Python 3.10:

```
    class ThreadingHTTPServer(socketserver.ThreadingMixIn, HTTPServer):
        daemon_threads = True
```

Исходный код (*https://github.com/python/cpython/blob/699ee016af5736ffc80f68359617611a22b72943/Lib/socketserver.py#L664*) класса `socketserver.ThreadingMixIn` насчитывает 38 строк, включая комментарии и строки документации. Пример 14.10 дает представление о его реализации.

Пример 14.10. Часть Lib/socketserver.py в Python 3.10

```
class ThreadingMixIn:
    """Класс-примесь для обработки каждого запроса в отдельном потоке."""

    # 8 строк опущено

    def process_request_thread(self, request, client_address): ❶
        ... # 6 строк опущено

    def process_request(self, request, client_address): ❷
        ... # 8 строк опущено

    def server_close(self): ❸
        super().server_close()
        self._threads.join()
```

❶ `process_request_thread` не вызывает `super()`, потому что это новый, а не переопределенный метод. Его реализация вызывает три метода экземпляра, которые `HTTPServer` предоставляет или наследует.

❷ Этот метод переопределяет метод `process_request`, который `HTTPServer` наследует от `socketserver.BaseServer`. При этом запускается поток, а фактическая работа делегируется методу `process_request_thread`, работающему в этом потоке. Функция `super()` не вызывается.

❸ `server_close` вызывает `super().server_close()`, чтобы прекратить прием запросов, а затем ждет завершения потоков, запущенных `process_request`.

Описание класса `ThreadingMixIn` находится в документации по модулю `socketserver` рядом с `ForkingMixin`. Последний предназначен для поддержки конкурентных серверов, основанных на `os.fork()`, API для запуска дочернего процесса в POSIX-совместимых системах типа Unix.

Примеси в обобщенных представлениях Django

Для чтения этого раздела не нужно быть знатоком Django. Я использую лишь малую часть данного каркаса как практический пример применения множественного наследования и постараюсь по ходу дела сообщить все необходимые сведения, предполагая, правда, что вы имеете какой-то опыт разработки серверных веб-приложений на другом языке или с помощью иного каркаса.

В Django представление – это вызываемый объект, который принимает в качестве аргумента объект, представляющий HTTP-запрос, и возвращает объект, представляющий HTTP-ответ. Нас в этом обсуждении будут интересовать различные ответы. Они могут быть совсем простыми, например ответ с перенаправлением, вообще не имеющий тела, или весьма сложными, например страница каталога интернет-магазина, которая строится по HTML-шаблону и содержит список товаров с кнопками для покупки и ссылками на страницы подробной информации.

Первоначально Django предоставлял набор функций, называемых обобщенными представлениями, которые реализовывали наиболее распространенные частные случаи. Например, на многих сайтах показываются результаты поиска, которые включают информацию о различных объектах, причем список может быть многостраничным, а для каждого объекта имеется ссылка на страницу с детальной информацией. В Django списковое представление и детальное представление спроектированы так, чтобы совместно решать эту задачу: списковое представление отображает результаты поиска, а детальное формирует страницы с информацией об отдельных объектах.

Однако изначально обобщенные представления были просто функциями, т. е. не допускали расширения. Если нужно было сделать что-то похожее на обобщенное списковое представление, но не в точности совпадающее с ним, приходилось начинать с нуля.

Концепция представлений на основе классов появилась в Django 1.3 вместе с набором классов обобщенных представлений, состоящим из базовых классов, примесей и готовых конкретных классов. В Django 3.2 базовые классы и примеси находятся в модуле `base` из пакета `django.views.generic` (рис. 14.3). В верхней части диаграммы мы видим два класса, на которые возложены совершенно разные обязанности: `View` и `TemplateResponseMixin`.

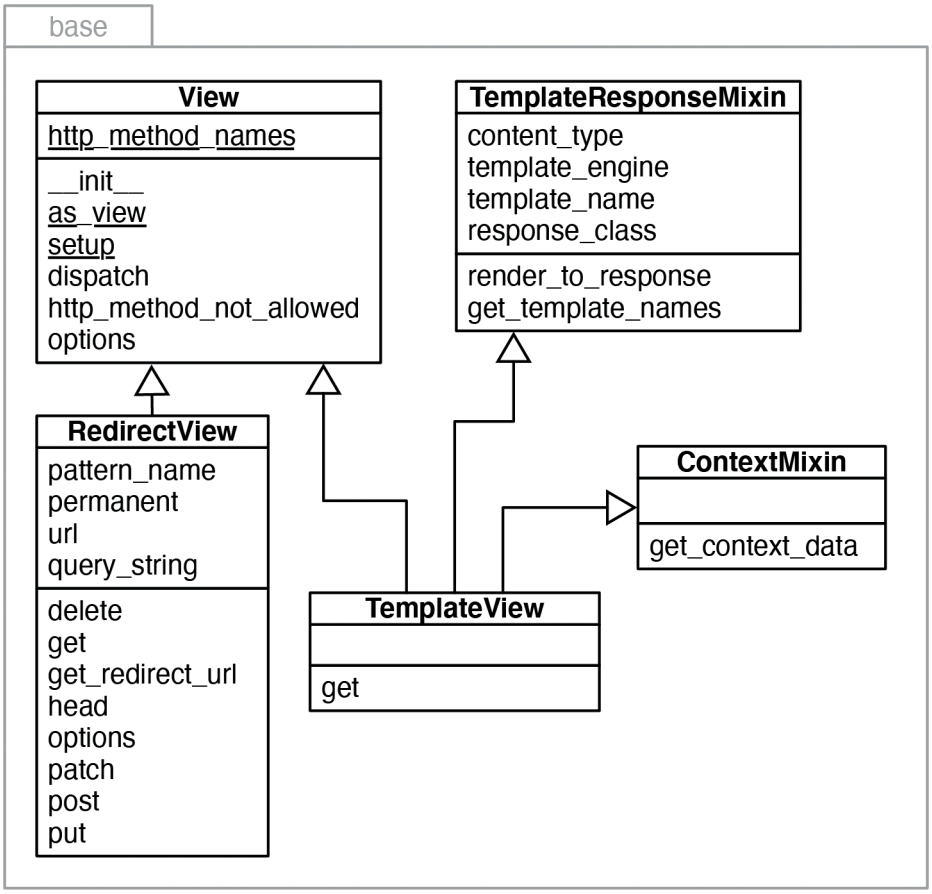


Рис. 14.3. UML-диаграмма классов из модуля `django.views.generic.base`

Замечательным ресурсом для изучения этих классов является сайт Classy Class-Based Views (*http://ccbv.co.uk/*), где организована удобная навигация и можно посмотреть все методы каждого класса (унаследованные, переопределенные и добавленные), диаграммы, документацию и даже перейти в исходный код на сайте GitHub (*https://github.com/django/django/tree/main/django/views/generic*).

`View` является базовым классом всех представлений (он мог бы быть абстрактным) и предоставляет основную функциональность, например метод `dispatch`, делегирующий работу методам-обработчикам – `get`, `head`, `post` и др., – которые реализованы в конкретных классах для обработки различных глаголов HTTP[1]. Класс `RedirectView` наследует только `View` и, как видите, реализует методы `get`, `head`, `post` и т. д.

[1] Программирующие на Django знают, что метод класса `as_view` – самая заметная часть интерфейса `View`, но нам это сейчас неинтересно.

Но если предполагается, что конкретные подклассы `View` реализуют методы-обработчики, то почему же они не являются частью интерфейса `View`? Причина проста: подклассы вольны реализовывать лишь те обработчики, которые считают нужным поддержать. Класс `TemplateView` служит только для отображения содержимого, поэтому реализует лишь метод `get`. Если объекту `TemplateView` будет послан POST-запрос, то унаследованный метод `View.dispatch` обнаружит, что обработчика `post` нет, и отправит HTTP-ответ `405 Method Not Allowed`[1].

Класс `TemplateResponseMixin` предоставляет функциональность, интересную только представлениям, нуждающимся в шаблоне. Но, например, у представления `RedirectView` нет тела, поэтому и шаблон ему не нужен, а значит, оно не наследует эту примесь. Примесь `TemplateResponseMixin` предоставляет набор поведений классу `TemplateView` и прочим представлениям, отрисовывающим шаблон, например `ListView` или `DetailView`, определенным в других модулях пакета `django.views.generic`. На рис. 14.4 показана диаграмма классов из модуля `django.views.generic.list` и частично из модуля `base`.

Для пользователей Django самым важным из показанных на рис. 12.5 классов является `ListView`; это агрегатный класс, в котором вообще нет кода (его тело не содержит ничего, кроме строки документации). У объекта класса `ListView` имеется атрибут экземпляра `object_list`, который шаблон может обойти, чтобы показать содержимое страницы; обычно это результат запроса к базе данных, содержащий несколько объектов. Вся функциональность, относящаяся к генерации этого итерируемого объекта, находится в примеси `MultipleObjectMixin`. Эта же примесь предоставляет сложную логику разбиения на страницы, необходимую для показа на одной странице части результатов и ссылок на другие страницы.

Предположим, что требуется создать представление, которое не отрисовывает шаблон, а порождает список объектов в формате JSON. Для этой цели существует класс `BaseListView`. Это точка расширения, которая объединяет функциональность классов `View` и `MultipleObjectMixin`, но без накладных расходов, обусловленных механизмом шаблонов.

Основанный на классах API представлений в Django – пример более правильного, чем в Tkinter, использования множественного наследования. В частности, разобраться в классах-примесях здесь очень просто: у каждого свое четко определенное назначение и имя, оканчивающееся суффиксом `_Mixin`.

[1] Знакомые с паттернами проектирования заметят, что механизм диспетчеризации в Django – динамический вариант паттерна Шаблонный метод (*http://en.wikipedia.org/wiki/Template_method_pattern*). Динамический – потому что класс `View` не заставляет свои подклассы реализовывать все обработчики, а `dispatch` на этапе выполнения проверяет, существует ли обработчик поступившего запроса.

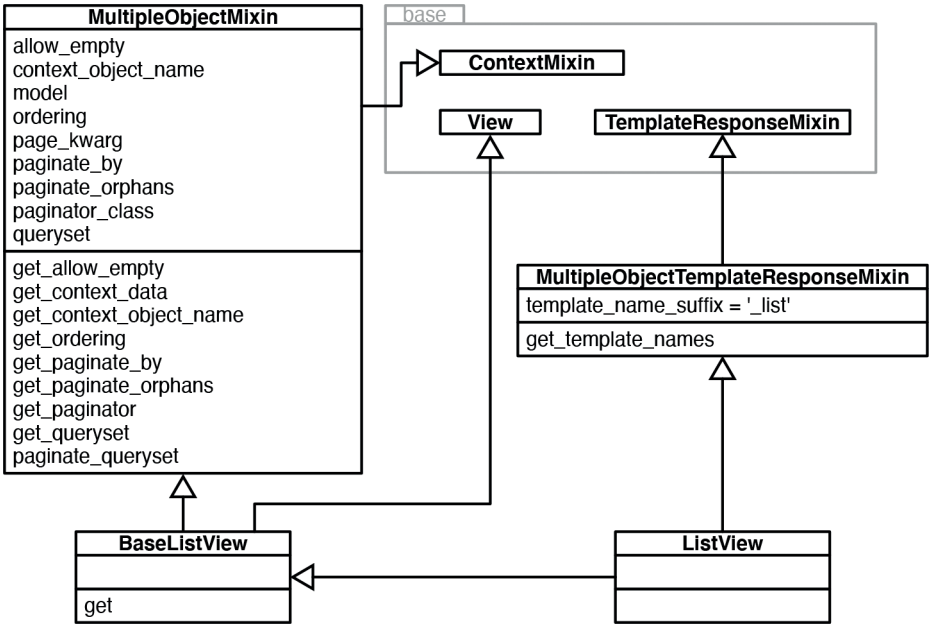


Рис. 14.4. UML-диаграмма классов из модуля `django.views.generic.list`. Все три класса из модуля `base` свернуты (см. рис. 14.3). В классе `ListView` нет ни методов, ни атрибутов, это агрегатный класс

Основанные на классах представления не все пользователи Django приняли на ура. Многие пользуются ими как черными ящиками, но если необходимо создать что-то новое, то по-прежнему пишут монолитные функции, которые берут на себя все обязанности, – вместо того чтобы попытаться повторно использовать классы представлений и примеси.

Чтобы в полной мере понять, как использовать представления, основанные на классах, и как расширять их для решения задач конкретного приложения, нужно время, но я пришел к выводу, что это время будет потрачено не зря: они позволяют устранить стереотипный код, упрощают повторное использование и даже улучшают взаимодействие между членами команды – например, за счет стандартизации имен шаблонов и переменных, передаваемых в контекст шаблона. Представления, основанные на классах, – это представления Django «on rails», как в Ruby.

Множественное наследование в Tkinter

Экстремальный пример множественного наследования в стандартной библиотеке Python дает пакет построения графических интерфейсов Tkinter (*https://docs.python.org/3/library/tkinter.html*). Я уже использовал иерархию одного виджета Tkinter для иллюстрации MRO на рис. 14.2, а на рис. 14.5 показаны все классы виджетов, присутствующие в базовом пакете `tkinter` (кроме них, есть еще много виджетов в подпакете `tkinter.ttk` – *https://docs.python.org/3/library/tkinter.ttk.html*).

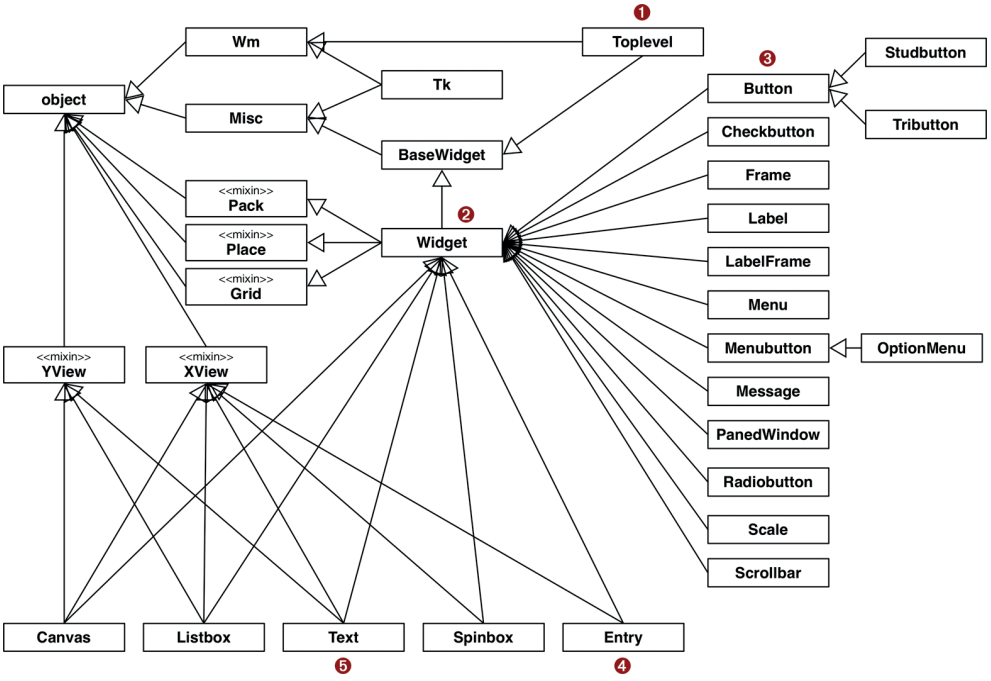


Рис. 14.5. Сводная UML-диаграмма иерархии классов Tkinter; классы со стереотипом «mixin» предоставляют конкретные методы другим классам с помощью множественного наследования

Когда я пишу эти строки, пакету Tkinter уже исполнилось 25 лет, и его нельзя считать примером лучших современных методик. Однако он показывает, как множественное наследование использовалось, когда кодировщики не придавали большого значения его недостаткам. И он послужит примером того, как делать не надо, когда в следующем разделе мы будем обсуждать рекомендуемые подходы.

Рассмотрим классы, показанные на рис. 14.5.

❶ `Toplevel`: класс окна верхнего уровня в приложении Tkinter.
❷ `Widget`: суперкласс всех видимых объектов, которые можно разместить в окне.
❸ `Button`: обычная кнопка.
❹ `Entry`: однострочное редактируемое текстовое поле.
❺ `Text`: многострочное редактируемое текстовое поле.

Вот как выглядят MRO этих классов, напечатанные функцией `print_mro` из примера 14.7:

```
>>> import tkinter
>>> print_mro(tkinter.Toplevel)
Toplevel, BaseWidget, Misc, Wm, object
>>> print_mro(tkinter.Widget)
Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Button)
Button, Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Entry)
```

```
Entry, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, object
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

По современным стандартам, иерархия классов в Tkinter чрезмерно глубокая. В немногих частях стандартной библиотеки Python встречается более трех-четырех уровней конкретных классов, и то же самое можно сказать о библиотеке классов Java. Однако интересно отметить, что некоторые из самых глубоких иерархий в библиотеке классов Java относятся как раз к пакетам, связанным с программированием GUI: `java.awt` (*https://docs.oracle.com/javase/10/docs/api/java/awt/package-tree.html*) и `javax.swing` (*https://docs.oracle.com/javase/10/docs/api/javax/swing/package-tree.html*). *Squeak* (*https://squeak.org/*), современная свободная версия Smalltalk, включает мощный инновационный инструментарий GUI под названием Morphic, и для него тоже характерна глубокая иерархия классов. По моему опыту, наборы инструментов для разработки GUI – та область, где наследование наиболее полезно.

Обратите внимание на то, как эти классы связаны друг с другом.

- `Toplevel` – единственный графический класс, не наследующий `Widget`, потому что это окно верхнего уровня, и оно не ведет себя как виджет, например его нельзя присоединить к окну или фрейму. `Toplevel` наследует классу `Wm`, который предоставляет функции прямого доступа к объемлющему оконному менеджеру, например для установки заголовка окна и настройки его рамки.
- `Widget` наследует непосредственно `BaseWidget`, а также классам `Pack`, `Place` и `Grid`. Последние три класса – менеджеры компоновки, они отвечают за расположение виджетов в окне или фрейме. Каждый инкапсулирует свою стратегию и API размещения виджетов.
- `Button`, как и большинство виджетов, напрямую наследует только `Widget`, а опосредованно – классу `Misc`, который предоставляет десятки методов каждому виджету.
- `Entry` является подклассом `Widget` и `XView` – класса, который реализует горизонтальную прокрутку.
- `Text` наследует `Widget`, `XView` и `YView` – классу, реализующему вертикальную прокрутку.

Далее мы обсудим некоторые рекомендации по использованию множественного наследования и посмотрим, согласуется ли с ними Tkinter.

Жизнь с множественным наследованием

То, что писал Алан Кэй в эпиграфе к этой главе, остается верным и по сию пору: по-прежнему не существует общей теории наследования, которая могла бы служить руководством к действию для программистов-практиков. То, что у нас есть, – не более чем набор эвристических правил, паттернов проектирования, «передовых практик», наукообразных акронимов, табу и т. д. Некоторые из них и вправду дают полезные рекомендации, но ни про одно нельзя сказать, что оно всеми принято или всегда применимо.

Применяя наследование, пусть даже и не множественное, легко получить запутанный и хрупкий дизайн. Ввиду отсутствия исчерпывающей теории приведем несколько советов, как избежать графов классов, напоминающих блюдо спагетти.

Предпочитайте композицию наследованию класса

Название этого раздела взято прямиком из книги «Паттерны проектирования»[1], и лучше совета не придумаешь. Освоив наследование, очень легко впасть в грех злоупотребления им. Организация объектов в симпатичную иерархию импонирует нашему чувству порядка; а программисты делают это просто забавы ради.

Отдавая предпочтение композиции, мы получаем более гибкий дизайн. Например, класс `tkinter.Widget` мог бы не наследовать методы от всех менеджеров компоновки, а хранить ссылку на менеджер и вызывать его методы. В конце концов, `Widget` же не должен «быть» менеджером компоновки, но мог бы пользоваться его услугами с помощью делегирования. Тогда было бы нетрудно добавить новый менеджер компоновки, не изменяя иерархию классов виджетов и не беспокоясь по поводу возможных конфликтов имен. Даже в случае одиночного наследования этот принцип повышает гибкость, поскольку создание подкласса – форма тесной связанности, а глубокие деревья наследования обычно оказываются хрупкими.

Композиция и делегирование могут заменить использование примесей, когда нужно предоставить некоторый набор поведений различным классам, но не могут заменить наследование интерфейсов как средство определения иерархии типов.

Разберитесь, зачем наследование используется в каждом конкретном случае

Имея дело с множественным наследованием, полезно ясно определить, по каким причинам вообще создается подкласс. Основные причины таковы:

- наследование интерфейса создает подтип, подразумевая связь «является». Для этой цели лучше использовать ABC;
- наследование реализации позволяет избежать дублирования кода. В этом случае могут помочь примеси.

На практике обе причины часто идут рука об руку, но если удается прояснить намерение, сделайте это. Наследование ради повторного использования кода – это деталь реализации, его нередко можно заменить композицией и делегированием. С другой стороны, наследование интерфейса – это становой хребет любого каркаса. При наследовании интерфейса следует по возможности использовать в качестве базовых классов только ABC.

Определяйте интерфейсы явно с помощью ABC

В современном Python класс, предназначенный для определения интерфейса, следует явно делать абстрактным базовым классом или подклассом `typing.Protocol`. ABC должен быть подклассом только `abc.ABC` или другого ABC. Множественное наследование ABC не приводит ни к каким проблемам.

[1] На стр. 20 введения.

Используйте примеси для повторного использования кода

Если класс предназначен для того, чтобы предоставлять реализации методов различным не связанным между собой подклассам, не подразумевая связи «является», то его следует явно делать *классом-примесью*. Концептуально примесь не определяет нового типа, а просто служит контейнером общеполезных методов. Примесь никогда не инстанцируется, и конкретные классы не должны ей наследовать. Каждая примесь должна определять четко очерченное поведение, реализуя несколько очень тесно связанных методов. В примесях желательно избегать внутреннего состояния, т. е. в классе-примеси не должно быть атрибутов экземпляра.

В Python нет формального способа сказать, что класс является примесью, поэтому настоятельно рекомендуется включать в имя суффикс `Mixin`.

Предоставляйте пользователям агрегатные классы

> Класс, который конструируется в основном путем наследования примесям и не добавляет собственной структуры или поведения, называется агрегатным классом.
>
> – Буч и др.[1]

Если какая-то комбинация ABC или примесей может быть особенно полезна в клиентском коде, предоставьте класс, который объединяет их разумным образом.

Вот, например, полный исходный код класса `ListView` из Django, показанного справа внизу на рис. 14.4 (*https://github.com/django/django/blob/b64db05b9cedd96905d637a2d824cbbf428e40e7/django/views/generic/list.py#L194*):

```
class ListView(MultipleObjectTemplateResponseMixin, BaseListView):
    """
    Отобразить список объектов, заданный с помощью `self.model` или
    `self.queryset`. `self.queryset` может быть любым итерируемым
    объектом, а не только queryset.
    """
```

Тело класса `ListView` пусто, но сам класс несет полезную функцию: объединяет примесь и базовый класс, которые должны использоваться вместе.

Еще один пример дает `tkinter.Widget`, который имеет четыре базовых класса и ни одного собственного метода или атрибута, только строку документации. Благодаря агрегатному классу `Widget` мы можем создать новый виджет с требуемыми примесями, не думая о том, в каком порядке их нужно перечислять в объявлении, чтобы все работало, как задумано.

Отметим, что агрегатные классы не обязаны быть пустыми, но часто именно так и бывает.

Наследуйте только классам, предназначенным для наследования

Рецензируя эту главу, Леонардо Рохаэль предложил включить следующее предупреждение:

---

[1] *Grady Booch et al.* Object-Oriented Analysis and Design with Applications. 3-е изд. Addison-Wesley. С. 109.

Наследование сложному классу и переопределение его методов – занятие, чреватое ошибками, потому что методы суперкласса могут неожиданно игнорировать переопределения, сделанные в подклассе. Старайтесь избегать переопределения методов или, по крайней мере, ограничьтесь наследованием классов, которые специально спроектированы так, чтобы их было легко расширять, и только так, как было предусмотрено при их проектировании.

Совет-то хороший, но как узнать, что класс проектировался для расширения?

Первый ответ – заглянуть в документацию (иногда в форме строк документации или даже комментариев в коде). Например, Python-пакет `socketserver` (*https://docs.python.org/3/library/socketserver.html*) описывается как «каркас для построения сетевых серверов». Входящий в него класс `BaseServer` (*https://docs.python.org/3/library/socketserver.html#socketserver.BaseServer*) предназначен для наследования, как следует из самого названия. Важнее, впрочем, то, что в документации и в строке документации в исходном коде явно указано, какие из его методов предназначены для переопределения в подклассах.

В Python ⩾ 3.8 есть новый способ сделать такие проектные ограничения явными, он описан в документе PEP 591 «Adding a final qualifier to typing» (*https://peps.python.org/pep-0591/*), где вводится декоратор `@final` (*https://docs.python.org/3/library/typing.html#typing.final*), который можно применять к классам и отдельным методам, в результате чего IDE или программа проверки типов сообщит о неразумных попытках унаследовать такому классу или переопределить такой метод[1].

Воздерживайтесь от наследования конкретным классам

Наследование конкретным классам опаснее наследования ABC и примесям, потому что у экземпляров конкретных классов обычно имеется внутреннее состояние, которое легко можно повредить при переопределении зависящих от него методов. Даже если ваши методы кооперативные, т. е. вызывают `super()`, и даже если их внутреннее состояние хранится в атрибутах, закрытых благодаря синтаксической конструкции `__x`, все равно не счесть возможностей внести ошибку вследствие переопределения методов.

Во вставном эссе «Водоплавающие птицы и ABC» в главе 13 Алекс Мартелли приводит цитату из книги Скотта Мейерса «Наиболее эффективное использование C++»[2], в которой высказано еще более радикальное мнение: «все нелистовые классы должны быть абстрактными». Иными словами, Мейер говорит, что наследовать можно только абстрактным классам.

Если вам все-таки необходимо использовать наследование ради повторного использования кода, то этот код следует размещать в подмешиваемых методах ABC или в явно поименованных классах-примесях.

Tkinter: хороший, плохой, злой

Модуль Tkinter не следует большинству изложенных выше рекомендаций, за исключением совета предоставлять пользователям агрегатные классы.

---

[1] В PEP 591 введена также аннотация `Final` (*https://docs.python.org/3/library/typing.html#typing.Final*) для переменных или атрибутов, которым не следует переприсваивать значение или переопределять.

[2] *Скотт Мейерс.* Наиболее эффективное использование C++. М.: ДМК Пресс, 2012 // *https://dmkpress.com/catalog/computer/programming/c/978-5-94074-990-5/*

Но даже в этом отношении я бы не стал ставить его в пример, потому что композиция, пожалуй, была бы уместнее для интеграции менеджеров компоновки с классом `Widget`, о чем было написано в рекомендации «Предпочитайте композицию наследованию класса».

Помните, что Tkinter является частью стандартной библиотеки еще со времен версии Python 1.1, выпущенной в 1994 году. Tkinter – это слой поверх великолепной библиотеки Tk, поставляемой вместе с языком Tcl. Комбинация Tcl/Tk изначально не была объектно-ориентированной, поэтому Tk API представляет собой просто обширный набор функций. Однако концептуально эта библиотека в высшей степени объектно-ориентированная, пусть даже реализация Tcl таковой не является.

Строка документации `tkinter.Widget` начинается словами «Internal class». Это наводит на мысль, что `Widget`, наверное, следовало бы сделать ABC. Хотя у класса `Widget` нет собственных методов, он тем не менее определяет интерфейс. Его посыл таков: «Можете рассчитывать, что каждый виджет Tkinter предоставляет основные методы виджета (`__init__`, `destroy` и десятки функций из Tk API) в дополнение к методам всех трех менеджеров компоновки». Можно согласиться, что такое определение интерфейса далеко от совершенства (слишком широкое), но все же это интерфейс, а `Widget` «определяет» его как объединение интерфейсов своих суперклассов.

Класс `Tk`, который инкапсулирует прикладную логику графического интерфейса пользователя (GUI), наследует классам `Wm` и `Misc`, не являющимся ни абстрактными, ни примесями (`Wm` – не совсем примесь, потому что ему наследует `TopLevel`). От самого имени класса `Misc` очень сильно отдает *запашком*. В `Misc` больше 100 методов, и ему наследуют все виджеты. А разве каждому виджету нужны методы для работы с буфером обмена, для выделения текста, для управления таймером и т. д.? Ведь невозможно вставить что-то в кнопку из буфера обмена или выделить текст полосы прокрутки. Класс `Misc` следовало бы разбить на несколько специализированных классов-примесей и не заставлять все виджеты наследовать каждому из этих классов.

Но будем справедливы – пользователю Tkinter вовсе необязательно знать о множественном наследовании. Эта деталь реализации скрыта за фасадом классов виджетов, которые вы инстанцируете или которым наследуете в своем коде. Однако пользователь почувствует последствия злоупотребления множественным наследованием, если наберет `dir(tkinter.Button)` и попытается найти нужный метод среди 214 перечисленных атрибутов. И вам придется столкнуться лицом к лицу со сложностью, если вы пожелаете реализовать новый виджет Tk.

Несмотря на все проблемы, Tkinter предлагает стабильный, гибкий и вполне современный на вид интерфейс, а если вы воспользуетесь пакетом `tkinter.ttk`, то получите виджеты, поддерживающие темы. Кроме того, некоторые оригинальные виджеты, например `Canvas` и `Text`, обладают поразительно богатыми возможностями. Добавив немного своего кода, вы можете превратить объект `Canvas` в простое приложение для рисования, поддерживающее перетаскивание. С Tkinter и Tcl/Tk определенно стоит познакомиться, если вы занимаетесь программированием GUI.

На этом мы завершаем экскурсию по лабиринту наследования.

Резюме

Мы начали эту главу с обзора функции `super()` в контексте одиночного наследования. Затем обсудили проблему наследования встроенным типам: их методы, реализованные на C, не вызывают методы, переопределенные в подклассах, за исключением немногих частных случаев. Именно поэтому в тех случаях, когда нам нужен специальный список, словарь или строка, проще наследовать не классам `list`, `dict` или `str`, а классам `UserList`, `UserDict` или `UserString` – все они определены в модуле `collections` (*https://docs.python.org/3/library/collections.html*) и фактически обертывают встроенные типы, делегируя им работу, – это три примера использования композиции вместо наследования в стандартной библиотеке. Если требуемое поведение очень сильно отличается от поведения встроенных классов, то, быть может, проще унаследовать подходящему ABC из модуля `collections.abc` (*https://docs.python.org/3/library/collections.abc.html*) и написать собственную реализацию.

Остаток главы был посвящен обоюдоострому мечу множественного наследования. Сначала мы познакомились с порядком разрешения методов, который закодирован в атрибуте класса `__mro__` и решает проблему потенциального конфликта имен в унаследованных методах. Мы также видели, как встроенная функция `super()` ведет себя, иногда неожиданно, в иерархиях с множественным наследованием. Функция `super()` спроектирована с целью поддержки классов-примесей, которые мы изучили на простом примере класса `UpperCaseMixin` для отображений, не зависящих от регистра.

Мы видели, как множественное наследование и подмешанные методы используются в абстрактных базовых классах, а также в примесях для исполнения запросов в отдельных потоках и процессах, входящих в состав модуля `socketserver`. Более сложные применения множественного наследования мы рассмотрели на примерах, основанных на классах представлений Django и набора инструментов Tkinter для разработки GUI. Tkinter не назовешь образцом современных передовых практик, это пример чрезмерно усложненных иерархий классов, которые можно встретить в унаследованных системах.

В заключение я предложил семь рекомендаций по использованию наследования и проиллюстрировал их на примере иерархии классов в Tkinter.

Отказ от наследования – даже одиночного – современная тенденция. Одним из самых успешных языков, созданных в XXI веке, является Go. В нем нет конструкции, именуемой «класс», но можно строить типы, представляющие собой структуры инкапсулированных полей, и присоединять к этим структурам методы. Go позволяет определять интерфейсы, которые проверяются компилятором с помощью структурной типизации, по аналогии со *статической утиной типизацией* – очень похоже на то, что мы теперь имеем в виде протокольных типов, начиная с версии Python 3.8. В Go имеется специальный синтаксис для построения типов и интерфейсов посредством композиции, но наследование он не поддерживает – даже наследование интерфейсов.

Таким образом, пожалуй, лучший совет по поводу наследования – избегайте его, если можете. Но зачастую просто нет выбора: применяемый каркас диктует свои правила.

Дополнительная литература

> Если говорить о чтении, то ясность и правильно исполненная композиция дадут сто очков вперед наследованию. Поскольку код гораздо чаще читают, чем пишут, избегайте подклассов вообще, но особенно сторонитесь смешения разных типов наследования и не пользуйтесь подклассами для организации разделения кода.
>
> – Хинек Шлавак, «Subclassing in Python Redux»

В последний раз перечитывая эту книгу, технический рецензент Юрген Гмах рекомендовал мне статью Хинека Шлавака «Subclassing in Python Redux» (*https://hynek.me/articles/python-subclassing-redux/*), из которой взята предыдущая цитата. Шлавак – автор пакета *attrs* и один из основных соавторов каркаса асинхронного программирования Twisted – проекта, запущенного Глифом Лефковицем в 2002 году. Со временем, по словам Шлавака, команда разработчиков ядра пришла к выводу, что слишком увлеклась наследованием. Его статья длинная, в ней много цитат из других статей и бесед. Всячески рекомендую.

В заключение к той статье Хинек Шлавак писал: «Не забывайте, что чаще всего вам достаточно просто функции». Я с ним согласен, и именно по этой причине в моей книге функции подробно рассматриваются раньше классов и наследования. Я ставил себе целью показать, сколь многого можно достичь с помощью функций, обращающихся к классам из стандартной библиотеки, не прибегая до поры к созданию собственных классов.

Наследование встроенным классам, функция `super` и такие продвинутые средства, как дескрипторы и метаклассы, – все это описано в статье Гвидо ван Россума «Unifying types and classes in Python 2.2» (*https://www.python.org/download/releases/2.2.3/descrintro/*). С тех пор никаких существенных изменений в эти вещи не вносилось. Python 2.2 стал потрясающей вехой в эволюции языка, добавившей несколько мощных новых средств в единое целое и не нарушившей при этом обратной совместимости. Все новые возможности по умолчанию были отключены. Чтобы воспользоваться ими, нужно было явно унаследовать `object` – прямо или косвенно – и создать «класс в новом стиле». В Python 3 любой класс наследует `object`.

В книге David Beazley, Brian K. Jones «*Python Cookbook*», 3-е издание (O'Reilly), есть несколько рецептов, демонстрирующих использование `super()` и классов-примесей. Можете начать с просветляющего раздела «8.7. Вызов метода родительского класса» (*https://www.oreilly.com/library/view/python-cookbook-3rd/9781449357337/ch08.html#super*) и следовать по ведущим из него внутренним ссылкам.

Раймонд Хэттингер в статье «Python's super() considered super!» (*https://rhettinger.wordpress.com/2011/05/26/super-considered-super/*) объясняет работу функции `super` и множественное наследование в Python с позитивной точки зрения. Она была написана в ответ на статью «Python's Super is nifty, but you can't use it» (известную также под названием «Python's Super Considered Harmful») (*https://fuhm.net/super-harmful/*) Джеймса Найта. Ответ Мартина Питерса на вопрос «How to use super() with one argument?» (*https://stackoverflow.com/questions/ 30190185/how-to-use-super-with-one-argument/30190341#30190341*) содержит краткое и глубокое объяснение `super`, включая ее связь с дескрипторами, изучение которых мы отложим до главы 23. Такова природа `super`. Она проста в базовых случаях

применения, но является мощным и сложным инструментом, затрагивающим самые потаенные динамические механизмы Python, которые редко встретишь в других языках.

Несмотря на заглавия этих статей, проблема не в самой встроенной функции `super` – которая в Python 3 не так безобразна, как в Python 2. Настоящая проблема – множественное наследование и присущие ему внутренние сложности. Мишель Симионато не ограничился критикой, а предложил решение в своей статье «Setting Multiple Inheritance Straight» (*https://www.artima.com/weblogs/viewpost.jsp?thread=246488*): он реализовал классы-характеристики (traits) – ограниченную форму примесей, впервые предложенную в языке Self. Перу Симионату принадлежит целая серия статей о множественном наследовании в Python, включая «The wonders of cooperative inheritance, or using super in Python 3» (*https://www.artima.com/weblogs/viewpost.jsp?thread=281127*), «Mixins considered harmful», часть 1 (*https://www.artima.com/weblogs/viewpost.jsp?thread=236275*) и часть 2 (*https://www.artima.com/weblogs/viewpost.jsp?thread=246483*), и «Things to Know About Python Super», часть 1 (*https://www.artima.com/weblogs/viewpost.jsp? thread=236275*), часть 2 (*https://www.artima.com/weblogs/viewpost.jsp? thread=236278*) и часть 3 (*https://www.artima.com/weblogs/viewpost.jsp?thread= 237121*). В ранних статьях используется синтаксис `super` из Python 2, но они по-прежнему актуальны.

Я читал первое издание книги Grady Booch et al. «Object-Oriented Analysis and Design», 3-е издание, и горячо рекомендую ее как общее введение в объектно-ориентированное мышление вне зависимости от языка программирования. Эта книга – редкий пример обсуждения множественного наследования без предрассудков.

Теперь больше, чем когда-либо прежде, стало модным избегать наследования, поэтому приведу две ссылки на источники, показывающие, как это сделать. Брэндон Родс написал статью «The Composition Over Inheritance Principle» (*https://python-patterns.guide/gang-of-four/composition-over-inheritance/*), вошедшую в состав его прекрасного руководства «Python Design Patterns». Оджи Факлер и Натаниэль Маниста провели презентацию «The End Of Object Inheritance & The Beginning Of A New Modularity» (*https://www.youtube.com/watch?v=3MNVP9-hglc*) на конференции PyCon 2013. Факлер и Маниста рассуждают об организации систем вокруг интерфейсов и функций, которые обрабатывают объекты, реализующие эти интерфейсы, избегая тесной связанности и ошибок, присущих классам и наследованию. Это сильно напоминает мне путь Go, но они выступают за его применение в Python.

Поговорим

Думайте, какие классы вам действительно необходимы

> Мы начали продвигать идею наследования как способ, который позволил бы начинающим положить в основу каркасы, которые было по силам спроектировать только экспертам.
>
> – Алан Кэй, «The Early History of Smalltalk»[1]

[1] *Alan Kay.* The Early History of Smalltalk. SIGPLAN Not. 28, 3 (March 1993), 69–95. Доступно также онлайн (*http://worrydream.com/EarlyHistoryOfSmalltalk/*). Спасибо моему другу Кристиано Андерсону, поделившемуся этой ссылкой.

Подавляющее большинство программистов пишут приложения, а не каркасы. Но даже те, кто разрабатывает каркасы, скорее всего, тратят значительную (если не основную) часть своего времени на создание приложений. При написании приложений мы обычно не разрабатываем иерархии классов. Как правило, мы пишем классы, наследующие ABC или другим классам, предоставляемым каркасом. Авторам приложений крайне редко приходится писать класс, выступающий в роли суперкласса. Почти всегда мы создаем листовые классы (расположенные в листьях дерева наследования).

Если, разрабатывая приложение, вы ловите себя на создании многоуровневой иерархии классов, то, скорее всего, имеет место что-то из перечисленного ниже.

- Вы изобретаете велосипед. Посмотрите, нет ли в библиотеке или каркасе компонентов, которые вы могли бы повторно использовать в своем приложении.
- Вы работаете с плохо спроектированным каркасом. Поищите альтернативу.

Вы чрезмерно усложняете задачу. Вспомните *принцип KISS*.

- Вам наскучило писать приложения, и вы решили создать новый каркас. Примите поздравления и пожелания успеха!

Может также случиться, что к вашей ситуации применимы все четыре пункта: вам надоела рутина, и вы решили изобрести новый велосипед, построив свой чрезмерно усложненный и плохо спроектированный каркас, который заставляет вас писать один класс за другим для решения тривиальных задач. Надеюсь, вы получаете от этого удовольствие или хотя бы эта работа оплачивается.

Неправильное поведение встроенных типов: ошибка или так задумано?

Встроенные типы `dict`, `list` и `str` – важнейшие структурные элементы самого языка Python, поэтому они должны работать быстро, иначе плохо будет всем. Поэтому в CPython принят ряд компромиссных решений, из-за которых встроенные методы игнорируют методы, переопределенные в подклассах, что можно считать некорректным поведением. Возможный выход из этой ситуации: завести две реализации каждого типа: «внутреннюю», оптимизированную для использования самим интерпретатором, и внешнюю, которую можно было бы расширять.

Но именно это мы и имеем: классы `UserDict`, `UserList` и `UserString` работают не так быстро, как встроенные, зато расширяются без проблем. Принятый в CPython прагматичный подход означает, что и мы в своих приложениях обычно используем оптимизированные реализации, которым трудно наследовать. Это имеет смысл, если принять во внимание, что не так уж часто нам нужны специальные списки, отображения или строки. Следует только помнить о том, чем мы жертвуем.

Наследование в разных языках

Алан Кэй придумал термин «объектно-ориентированный», и в языке Smalltalk было только одиночное наследование, хотя существуют клоны с различными формами поддержки множественного наследования, в частности современные диалекты Squeak и Pharo Smalltalk, в которых поддерживаются характеристики (traits) – конструкции, играющие роль класса-примеси, но позволяющие избежать некоторых проблем множественного наследования.

Первым популярным языком с поддержкой множественного наследования стал C++, но это средство использовалось во вред настолько часто, что проектировщики языка Java – задуманного как замена C++ – отказались от множественного наследования реализации (т. е. от классов-примесей). И так было до выпуска Java 8, где появились методы по умолчанию, благодаря которым интерфейсы Java стали очень напоминать абстрактные классы, применяемые для определения интерфейсов в C++ и Python. После Java, пожалуй, самым распространенным языком на платформе JVM является Scala, и в нем реализованы характеристики.

Среди других языков, поддерживающих характеристики, упомянем последние стабильные версии PHP и Groovy, а также находящиеся в процессе разработки Rust и Raku – язык, который раньше был известен как Perl 6[1]. Так что будет справедливо сказать, что в 2021 году классы-характеристики – модная тенденция.

В Ruby принят оригинальный подход к множественному наследованию: оно не поддерживается, зато примеси являются полноправным языковым средством. Класс Ruby может включать модуль, так что определенные в модуле методы становятся частью реализации класса. Это «чистая» форма примеси, не нуждающаяся ни в каком наследовании, и ясно, что примесь в Ruby никак не влияет на тип класса, в котором используется. Тем самым мы получаем все преимущества примесей без многих связанных с ними проблем.

Два недавно созданных языка, привлекающих всеобщее внимание, – Go и Julia – серьезно ограничили наследование. Оба предназначены для программирования «объектов» и поддерживают полиморфизм (*https://en.wikipedia.org/wiki/ Polymorphism_(computer_science)*), но термина «класс» избегают.

В Go наследования нет вообще. В Julia иерархии типов существуют, однако подтип может наследовать только поведение, но не структуру, причем подтипы могут существовать только у абстрактных типов. Кроме того, методы в Julia реализованы с применением множественной диспетчеризации – более развитой формы механизма, который мы обсуждали в разделе «Обобщенные функции с одиночной диспетчеризацией» главы 9.

[1] Мой друг и технический рецензент Леонардо Рохаэль объяснит лучше меня: «То, что Perl 6 продолжает где-то маячить, но никак не явит себя миру, положило конец эволюции самого Perl. Теперь разработка Perl продолжается как отдельного языка (сейчас это уже версия 5.34), и нет никаких признаков, что этому будет положен конец в связи с появлением языка, который когда-то назывался Perl 6».

Глава 15

Еще об аннотациях типов

> Дорогой ценой я выучил урок: для небольших программ динамическая типизация – благо. Но для более крупных программ необходим более дисциплинированный подход. И хорошо, когда язык предлагает такую дисциплину, а не говорит «Да делай ты что хочешь».
>
> – Гвидо ван Россум, фанат Монти Пайтон[1]

Эта глава – продолжение главы 8, в ней излагаются дополнительные сведения о системе постепенной типизации в Python. Рассматриваются следующие темы:

- перегруженные сигнатуры функций;
- использование `typing.TypedDict` для аннотирования словарей, используемых как записи;
- приведение типов;
- доступ к аннотациям типов во время выполнения;
- обобщенные типы:
  - объявление обобщенного класса;
  - вариантность: инвариантные, ковариантные и контравариантные типы;
  - обобщенные статические протоколы.

Что нового в этой главе

Это новая глава, написанная специально для второго издания книги. Мы начнем с перегрузки.

Перегруженные сигнатуры

Функции в Python могут принимать различные комбинации аргументов. Декоратор `@typing.overload` позволяет аннотировать эти комбинации. Это особенно важно, когда тип возвращаемого функцией значения зависит от типа двух или более параметров.

Рассмотрим встроенную функцию `sum`. Ниже приведен текст справки `help(sum)`, для удобства читателя переведенный на русский язык:

---

[1] Из видео на YouTube «Интервью с создателями языков: Гвидо ван Россум, Джеймс Гослинг, Ларри Уолл и Андерс Хейлсберг», трансляция от 2 апреля 2019. Цитата, начинающаяся в 1:32:05, немного сокращена. Полную запись беседы см. по адресу *https://github.com/fluentpython/language-creators*.

```
>>> help(sum)
sum(iterable, /, start=0)
    Вернуть сумму значения 'start' (по умолчанию 0) и чисел, составляющих
    итерируемый объект iterable.

    Если объект iterable пуст, вернуть начальное значение.
    Эта функция предназначена для работы с числовыми значениями и может
    отвергать нечисловые типы.
```

Встроенная функция `sum` написана на C, но в *typeshed* есть для нее перегруженные аннотации типов, в файле *builtins.pyi* (*https://github.com/python/typeshed/blob/a8834fc d46339e17fc8add82b5803a1ce53d3d60/stdlib/2and3/builtins.pyi#L1434*):

```
@overload
def sum(__iterable: Iterable[_T]) -> Union[_T, int]: ...
@overload
def sum(__iterable: Iterable[_T], start: _S) -> Union[_T, _S]:
...
```

Сначала рассмотрим общий синтаксис перегрузки. Показанный выше код – все, что написано о `sum` в файле-заглушке (с расширением *.pyi*). Реализация находится в другом файле. Единственное назначение многоточия (...) – удовлетворить требование, предъявляемое к синтаксису тела функции, по аналогии с `pass`. Таким образом, *pyi*-файлы – допустимые Python-файлы.

В разделе «Аннотирование чисто позиционных и вариадических параметров» главы 8 отмечалось, что два начальных знака подчеркивания в `__iterable` – это описанное в документе PEP 484 соглашение о чисто позиционных параметрах, проверяемое Муру. Оно означает, что можно писать `sum(my_list)`, но не `sum(__iterable = my_list)`.

Средство проверки типов пытается сопоставить переданные аргументы с каждой перегруженной сигнатурой в порядке их перечисления. Вызов `sum(range(100), 1000)` не соответствует первой сигнатуре, потому что у нее всего один параметр. Но второй сигнатуре он соответствует.

Можно также использовать `@overload` в обычном Python-модуле, для чего следует записывать перегруженные сигнатуры прямо перед фактической сигнатурой и реализацией функции. В примере 15.1 показано, как могла бы выглядеть аннотированная реализация `sum` в Python-модуле.

Пример 15.1. mysum.py: определение функции `sum` с перегруженными сигнатурами

```
import operator
from collections.abc import Iterable
from typing import overload, Union, TypeVar

T = TypeVar('T')
S = TypeVar('S') ❶

@overload
def sum(it: Iterable[T]) -> Union[T, int]: ... ❷
@overload
def sum(it: Iterable[T], /, start: S) -> Union[T, S]: ... ❸
def sum(it, /, start=0): ❹
return functools.reduce(operator.add, it, start)
```

❶ Эта вторая `TypeVar` понадобится во второй перегруженной сигнатуре.
❷ Это сигнатура для простого случая: `sum(my_iterable)`. Результирующим типом может быть `T` – тип элементов, отдаваемых `my_iterable`, – или `int`, если итерируемый объект пуст, потому что значение параметра `start` по умолчанию равно `0`.
❸ Если `start` задано, то оно может иметь тип `S`, так что результирующим типом является `Union[T, S]`. Именно поэтому нам и нужна переменная `S`. Если бы мы повторно использовали `T`, то тип `start` должен был бы быть таким же, как тип элементов `Iterable[T]`.
❹ В сигнатуре фактической реализации функции нет аннотаций типов.

Довольно много строк для аннотации однострочной функции. Да, я знаю, это перебор. Но, по крайней мере, это была не функция `foo`.

Если вы хотите больше узнать о `@overload`, читая код, то в *typeshed* есть сотни примеров. На момент, когда я пишу этот текст, в файле-заглушке (*https://github.com/ python/typeshed/blob/a8834fcd46339e17fc8add82b5803a1ce53d3d60/stdlib/2and3/builtins.pyi*) для встроенных функций Python было 186 перегруженных сигнатур – больше, чем в любом другом файле-заглушке для стандартной библиотеки.

Используйте постепенную типизацию с пользой

Стремление получить стопроцентно аннотированный код может привести к тому, что аннотации типов будут только вносить шум, принося мало пользы. Рефакторинг с целью упростить аннотации типов может привести к громоздким API. Иногда лучше проявить разумный прагматизм и оставить часть кода неаннотированной.

Удобные API, которые мы называем питоническими, часто бывает трудно аннотировать. В следующем разделе приведен пример: для правильного аннотирования гибкой встроенной функции `max` необходимо шесть перегруженных вариантов.

Перегрузка max

Трудно добавлять аннотации типов к функциям, которые в полной мере задействуют мощные динамические средства Python.

Изучая *typeshed*, я наткнулся на отчет об ошибке #4051 (*https://github.com/python/typeshed/issues/4051*): Муру не предупреждает, что недопустимо передавать `None` в качестве одного из аргументов встроенной функции `max()` или передавать итерируемый объект, который в какой-то момент отдает значение `None`. В обоих случаях дело кончится исключением во время выполнения:

```
TypeError: '>' not supported between instances of 'int' and 'NoneType'
```

Документация по `max` начинается таким предложением:

> Вернуть наибольший элемент итерируемого объекта или наибольший из двух или более аргументов.

Лично мне такое описание кажется интуитивно понятным.

Но если передо мной стоит задача аннотировать описанную таким образом функцию, то следует спросить: какую именно? С итерируемым объектом или с двумя или более аргументами?

Реальность еще сложнее, потому что `max` принимает два факультативных именованных аргумента: `key` и `default`.

Я написал `max` на Python, чтобы было проще увидеть связь между ее работой и перегруженными аннотациями (встроенная функция `max` написана на C); см. пример 15.2.

Пример 15.2. mymax.py: функция `max`, переписанная на Python

```
# предложения импорта и определения опущены, см. следующий листинг

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

# перегруженные аннотации типов опущены, см. следующий листинг

def max(first, *args, key=None, default=MISSING):
    if args:
        series = args
        candidate = first
    else:
        series = iter(first)
        try:
            candidate = next(series)
        except StopIteration:
            if default is not MISSING:
                return default
            raise ValueError(EMPTY_MSG) from None
    if key is None:
        for current in series:
            if candidate < current:
                candidate = current
    else:
       candidate_key = key(candidate)
        for current in series:
            current_key = key(current)
            if candidate_key < current_key:
                candidate = current
                candidate_key = current_key
    return candidate
```

В этом примере нам важна не логика `max`, поэтому я не буду тратить время на описание реализации, а поясню только назначение `MISSING`. Константа `MISSING` – это уникальный объект, используемый как специальный маркер. Это значение по умолчанию именованного аргумента `default=`, поэтому `max` может принимать `default=None` и при этом различать следующие две ситуации.

1. Пользователь не задал значение аргумента `default=`, поэтому оно равно `MISSING`, и `max` возбуждает исключение `ValueError`, если `first` – пустой итерируемый объект.
2. Пользователь задал значение `default=`, быть может `None`, поэтому `max` возвращает это значение, если `first` – пустой итерируемый объект.

Чтобы исправить проблему #4051 (*https://github.com/python/typeshed/issues/4051*), я написал код, показанный в примере 15.3[1].

[1] Спасибо Джелле Зийлстра – одному из сопровождающих typeshed, – научившему меня нескольким вещам, в т. ч. как сократить число перегруженных сигнатур с девяти в моем оригинальном коде до шести.

Пример 15.3. mymax.py: начало модуля, содержащее предложения импорта, определения и перегруженные сигнатуры

```
from collections.abc import Callable, Iterable
from typing import Protocol, Any, TypeVar, overload, Union

class SupportsLessThan(Protocol):
    def __lt__(self, other: Any) -> bool: ...

T = TypeVar('T')
LT = TypeVar('LT', bound=SupportsLessThan)
DT = TypeVar('DT')

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

@overload
def max(__arg1: LT, __arg2: LT, *args: LT, key: None = ...) -> LT:
    ...
@overload
def max(__arg1: T, __arg2: T, *args: T, key: Callable[[T], LT]) -> T:
    ...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
    ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
    ...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...,
default: DT) -> Union[LT, DT]:
    ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
        default: DT) -> Union[T, DT]:
    ...
```

Длина моей реализации `max` на Python примерно такая же, как длина всех этих предложений импорта и объявлений. Благодаря утиной типизации в моем коде нет проверок `isinstance`, а проверку ошибок он обеспечивает такую же, как аннотации типов, – но, естественно, во время выполнения.

Главное преимущество `@overload` – максимально точное объявление типа возвращаемого значения в соответствии с типами переданных аргументов. Мы увидим, как проявляется это преимущество, когда будем рассматривать перегруженные варианты `max` группами по одному или по два.

Аргументы, реализующие SupportsLessThan, но без задания key и default

```
    @overload
    def max(__arg1: LT, __arg2: LT, *_args: LT, key: None = ...) -> LT:
        ...
    # ... строки опущены ...
    @overload
    def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
        ...
```

В этих случаях входами являются либо отдельные аргументы типа `LT`, реализующие протокол `SupportsLessThan`, либо итерируемый объект `Iterable`, отдающий такие элементы. Тип значения, возвращаемого `max`, такой же, как у фактических аргументов или элементов, как мы видели в разделе «Связанный TypeVar» главы 8.

Вот примеры вызовов, соответствующих этим перегруженным сигнатурам:

```
max(1, 2, -3) # возвращает 2
max(['Go', 'Python', 'Rust']) # возвращает 'Rust'
```

Аргумент key задан, аргумент default нет

```
@overload
def max(__arg1: T, __arg2: T, *_args: T, key: Callable[[T], LT]) -> T:
    ...
# ... строки опущены ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
    ...
```

Входами могут быть отдельные элементы любого типа `T` или один аргумент типа `Iterable[T]`, а `key=` должен быть вызываемым объектом, который принимает аргумент того же типа `T` и возвращает значение, реализующее протокол `SupportsLessThan`. Тип значения, возвращаемого `max`, такой же, как тип фактических аргументов.

Примеры вызовов, соответствующих этим перегруженным сигнатурам:

```
max(1, 2, -3, key=abs) # возвращает -3
max(['Go', 'Python', 'Rust'], key=len) # возвращает 'Python'
```

Аргумент default задан, аргумент key нет

```
@overload
def max(__iterable: Iterable[LT], *, key: None = ...,
        default: DT) -> Union[LT, DT]:
    ...
```

Входом может быть итерируемый объект, отдающий элементы типа `LT`, реализующего протокол `SupportsLessThan`. Тип значения, возвращаемого `max`, такой же, как тип фактических аргументов. Аргумент `default=` – это значение, возвращаемое, когда объект `Iterable` пуст. Следовательно, значение, возвращаемое `max`, должно иметь тип, являющийся объединением (`Union`) типа `LT` и типа аргумента `default`.

Примеры вызовов, соответствующих этой перегруженной сигнатуре:

```
max([1, 2, -3], default=0) # возвращает 2
max([], default=None) # возвращает None
```

Аргументы key и default заданы

```
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
        default: DT) -> Union[T, DT]:
    ...
```

Входами являются:

- объект `Iterable`, отдающий элементы любого типа `T`;
- вызываемый объект, который принимает аргумент типа `T` и возвращает значение типа `LT`, реализующего протокол `SupportsLessThan`;
- значение по умолчанию любого типа `DT`.

Значение, возвращаемое `max`, должно иметь тип, являющийся объединением типа `T` и типа аргумента `default`:

```
max([1, 2, -3], key=abs, default=None) # возвращает -3
max([], key=abs, default=None) # возвращает None
```

Уроки перегрузки max

Аннотации типов позволяют Муру выдавать о вызовах вида `max([None, None])` такое сообщение об ошибке:

```
mymax_demo.py:109: error: Value of type variable «_LT» of «max» cannot be «None»
```

С другой стороны, необходимость писать так много строк кода, чтобы помочь средству проверки типов, у многих отбивает желание программировать удобные и гибкие функции наподобие `max`. Если бы мне пришлось заново изобретать еще и функцию `min`, я мог бы повторно использовать значительную часть реализации `max`. Но я был бы вынужден скопировать все перегруженные объявления, пусть даже для `min` они не отличаются ничем, кроме имени функции.

Мой друг Жоао С. О. Буэно – один из самых талантливых разработчиков на Python, которых я знаю, – написал в Твиттере такое сообщение:

> Хотя выразить сигнатуру `max` трудно, в голове она укладывается очень легко. На мой взгляд, выразительность аннотаций очень ограничена по сравнению с выразительностью самого Python.

Теперь давайте изучим конструкцию `TypedDict`. Она не так полезна, как мне казалось поначалу, но кое-какие применения у нее есть. Эксперименты с `TypedDict` демонстрируют ограничения статической типизации для обработки динамических структур, таких как данные в формате JSON.

TypedDict

Возникает искушение использовать `TypedDict` для защиты от ошибок при обработке таких динамических структур данных, как ответы от JSON API. Но приведенные ниже примеры ясно показывают, что корректность обработки JSON-данных должна обеспечиваться во время выполнения, а не путем статической проверки типов. Если хотите проверять JSON-подобные структуры во время выполнения с помощью аннотаций типов, поинтересуйтесь пакетом *pydantic* (*https://pypi.org/project/pydantic/*) в архиве PyPI.

Словари Python иногда используются как записи, в которых ключами являются имена полей, а их значения могут иметь разные типы.

Например, рассмотрим запись, описывающую книгу, в формате JSON или на Python:

```
{"isbn": "0134757599",
 "title": "Refactoring, 2e",
 "authors": ["Martin Fowler", "Kent Beck"],
 "pagecount": 478}
```

До версии Python 3.8 не существовало хорошего способа аннотировать такую запись, потому что типы отображений, которые мы видели в разделе «Обобщенные отображения» главы 8, налагают ограничение: все значения должны быть одного типа.

Вот две робкие попытки аннотировать запись, похожую на показанный выше JSON-объект:

```
Dict[str, Any]
```

Значения могут быть любого типа.

```
Dict[str, Union[str, int, List[str]]]
```

Читается с трудом и не сохраняет связь между именами полей и соответствующими им типами: предполагается, что `title` должно иметь тип `str`, а не `int` и не `List[str]`.

Эта проблема решена в документе PEP 589 «TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys» (*https://peps.python.org/pep-0589/*). В примере 15.4 показан простой типизированный словарь `TypedDict`.

Пример 15.4. books.py: определение `BookDict`

```
from typing import TypedDict

class BookDict(TypedDict):
    isbn: str
    title: str
    authors: list[str]
    pagecount: int
```

На первый взгляд может показаться, что `typing.TypedDict` похож на построитель класса данных наподобие `typing.NamedTuple` (см. главу 5).

Но синтаксическое сходство обманчиво. `TypedDict` предназначен совершенно для другой цели. Он лишь поддерживает средства проверки типов и полностью игнорируется во время выполнения.

`TypedDict` предлагает две вещи:

- напоминающий класс синтаксис для аннотирования словаря типами значений каждого «поля»;
- конструктор, который говорит средству проверки типов, что оно должно ожидать словаря с указанными ключами и значениями.

На этапе выполнения конструктор типизированного словаря, например `BookDict`, играет роль плацебо: делает то же самое, что вызов конструктора `dict` с теми же аргументами.

Тот факт, что `BookDict` создает простой словарь `dict`, означает также, что:

- «поля» в определении псевдокласса не создают атрибутов экземпляра;
- нельзя написать инициализаторы со значениями по умолчанию для «полей»;
- определения методов не допускаются.

Пример 15.5. Использование `BookDict` не совсем так, как задумано

```
>>> from books import BookDict
>>> pp = BookDict(title='Programming Pearls', ❶
...               authors='Jon Bentley', ❷
...               isbn='0201657880',
...               pagecount=256)
>>> pp ❸
{'title': 'Programming Pearls', 'authors': 'Jon Bentley', 'isbn': '0201657880',
'pagecount': 256}
>>> type(pp)
<class 'dict'>
>>> pp.title ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'title'
>>> pp['title']
'Programming Pearls'
>>> BookDict.__annotations__ ❺
{'isbn': <class 'str'>, 'title': <class 'str'>, 'authors': typing.List[str],
 'pagecount': <class 'int'>}
```

❶ Мы можем вызывать `BookDict` как конструктор `dict` с именованными аргументами или передав аргумент типа `dict`, в т. ч. и литерал.
❷ Ой... Я забыл, что `authors` принимает список. Но постепенная типизация означает, что во время выполнения никакой проверки типов не производится.
❸ Результатом вызова `BookDict` является простой `dict`...
❹ ... поэтому мы не можем читать данные с помощью нотации `object.field`.
❺ Аннотации типов находятся в `BookDict.__annotations__`, а не в `pp`.

Без средства проверки типов `TypedDict` не полезнее комментариев: он помогает читать код, но не более того. Напротив, построители классов из главы 5 полезны, даже если мы не пользуемся средствами проверки типов, потому что во время выполнения они генерируют или дополняют пользовательский класс, экземпляр которого мы можем создать. Они также предоставляют полезные методы и функции, перечисленные в табл. 5.1.

В примере 15.6 строится допустимый `BookDict` и делается попытка применить к нему некоторые операции. Это показывает, как `TypedDict` помогает Муру обнаруживать ошибки (см. пример 15.7).

Пример 15.6. demo_books.py: допустимые и недопустимые операции над `BookDict`

```
from books import BookDict
from typing import TYPE_CHECKING

def demo() -> None: ❶
    book = BookDict( ❷
        isbn='0134757599',
        title='Refactoring, 2e',
        authors=['Martin Fowler', 'Kent Beck'],
        pagecount=478
    )
    authors = book['authors'] ❸
    if TYPE_CHECKING: ❹
        reveal_type(authors)❺
```

```
    authors = 'Bob' ❻
    book['weight'] = 4.2
    del book['title']

if __name__ == '__main__':
    demo()
```

❶ Не забывайте добавить тип возвращаемого значения, иначе Муру проигнорирует эту функцию.
❷ Это допустимый `BookDict`: все ключи присутствуют, а значения имеют правильный тип.
❸ Муру выведет тип `authors` из аннотации ключа `'authors'` в `BookDict`.
❹ Константа `typing.TYPE_CHECKING` равна `True`, только когда работает средство проверки типов. На этапе выполнения она всегда равна `False`.
❺ Условие в предыдущем предложении `if` препятствует вызову `reveal_type(authors)` во время выполнения. `reveal_type` – не функция Python, а отладочное средство, предоставляемое Муру. Именно поэтому она нигде не импортируется. Ее вывод показан в примере 15.7.
❻ Последние три строки функции `demo` некорректны. Они приводят к ошибкам, показанным в примере 15.7.

Проверка типов в скрипте *demo_books.py* из примера 15.6 дает результат, показанный в примере 15.7.

Пример 15.7. Результат применения средства проверки типов к demo_books.py

```
.../typeddict/ $ mypy demo_books.py
demo_books.py:13: note: Revealed type is 'built-ins.list[builtins.str]' ❶
demo_books.py:14: error: Incompatible types in assignment
                  (expression has type "str", variable has type "List[str]") ❷
demo_books.py:15: error: TypedDict "BookDict" has no key 'weight' ❸
demo_books.py:16: error: Key 'title' of TypedDict "BookDict" cannot be deleted ❹
Found 3 errors in 1 file (checked 1 source file)
```

❶ Это замечание – результат `reveal_type(authors)`.
❷ Тип переменной `authors` был выведен из типа выражения `book['authors']`, которым оно инициализировано. Нельзя присвоить `str` переменной типа `List[str]`. Программы проверки типов обычно возражают против изменения типа переменной[1].
❸ Нельзя присвоить значение ключу, отсутствующему в определении `BookDict`.
❹ Нельзя удалить ключ, присутствующий в определении `BookDict`.

Теперь посмотрим, как `BookDict` используется в сигнатурах функций для проверки типов при вызове функции.

Пусть требуется сгенерировать XML-код по записям о книгах, например:

[1] По состоянию на май 2020 года pytype это разрешает. Но в FAQ сказано, что в будущем будет запрещено. См. вопрос «Why didn't pytype catch that I changed the type of an annotated variable?» (Почему pytype не заметила, что я изменил тип аннотированной переменной) в pytype FAQ (*https://google.github.io/pytype/faq.html*).

```
<BOOK>
  <ISBN>0134757599</ISBN>
  <TITLE>Refactoring, 2e</TITLE>
  <AUTHOR>Martin Fowler</AUTHOR>
  <AUTHOR>Kent Beck</AUTHOR>
  <PAGECOUNT>478</PAGECOUNT>
</BOOK>
```

Если бы мы разрабатывали код на языке MicroPython, предназначенном для исполнения крохотным микроконтроллером, то могли бы написать функцию вроде той, что показана в примере 15.8[1].

Пример 15.8. books.py: функция `to_xml`

```
AUTHOR_ELEMENT = '<AUTHOR>{}</AUTHOR>'

def to_xml(book: BookDict) -> str: ❶
    elements: list[str] = [] ❷
    for key, value in book.items():
        if isinstance(value, list): ❸
            elements.extend(
                AUTHOR_ELEMENT.format(n) for n in value) ❹
        else:
            tag = key.upper()
            elements.append(f'<{tag}>{value}</{tag}>')
    xml = '\n\t'.join(elements)
    return f'<BOOK>\n\t{xml}\n</BOOK>'
```

❶ Весь смысл этого примера: использование `BookDict` в сигнатуре функции.

❷ Часто бывает необходимо аннотировать коллекции, которые вначале пусты, иначе Муру не сможет вывести тип элементов[2].

❸ Муру понимает проверки с помощью `isinstance` и в этом блоке рассматривает `value` как `list`.

❹ Когда я использовал `key == 'authors'` в качестве условия `if`, проверяющего вход в этот блок, Муру обнаружила ошибку в этой строке: `«object» has no attribute «__iter__»`, т. к. вывела, что `value`, возвращенное методом `book.items()`, имеет тип `object`, а этот тип не поддерживает метод `__iter__`, необходимый генераторному выражению. Если проверка производится с помощью `isinstance`, то все работает, потому что Муру знает, что в этом блоке `value` имеет тип `list`.

В примере 15.9 показана функция, которая разбирает строку в формате JSON типа `str` и возвращает `BookDict`.

Пример 15.9. books_any.py: функция `from_json`

```
def from_json(data: str) -> BookDict:
    whatever = json.loads(data) ❶
    return whatever ❷
```

[1] Я предпочитаю использовать пакет lxml (*https://lxml.de/*) для генерирования и разбора XML: с ним легко начать работу, он достаточно функциональный и быстрый. К сожалению, lxml и собственное дерево элементов Python *ElementTree* (*https://docs.python.org/3/library/xml.etree.elementtree.html*) не помещаются в ограниченную память моего гипотетического микроконтроллера.

[2] В документации по Муру этот вопрос обсуждается на странице «Типичные проблемы и решения» в разделе «Типы пустых коллекций».

❶ Значение, возвращаемое `json.loads()`, имеет тип `Any`[1].
❷ Я могу вернуть `whatever` – типа `Any`, – потому что `Any` *совместим с* любым типом, в т. ч. указанным в объявлении возвращаемого значения типом `BookDict`.

Второй момент примера 15.9 тоже очень важно иметь в виду: Муру не сообщает об ошибке в этом коде, но во время выполнения значение переменной `whatever` может быть не согласовано со структурой `BookDict`, да и вообще быть не словарем!

Если запустить Муру с флагом `--disallow-any-expr`, то она будет ругаться на две строки в теле `from_json`:

```
…/typeddict/ $ mypy books_any.py --disallow-any-expr
books_any.py:30: error: Expression has type "Any"
books_any.py:31: error: Expression has type "Any"
Found 2 errors in 1 file (checked 1 source file)
```

Упомянутые в этом фрагменте строки 30 и 31 – тело функции `from_json`. Мы можем подавить ошибку типизации, добавив аннотацию типа в инициализацию переменной `whatever`, как показано в примере 15.10.

Пример 15.10. books.py: функция `from_json` с аннотацией переменной

```
def from_json(data: str) -> BookDict:
    whatever: BookDict = json.loads(data) ❶
    return whatever ❷
```

❶ Наличие флага `--disallow-any-expr` не вызывает ошибок, когда выражение типа `Any` присваивается переменной с аннотацией типа.
❷ Теперь `whatever` имеет тип `BookDict` – тот, что объявлен для возвращаемого значения.

Пусть вас не убаюкивает ложное чувство типобезопасности в примере 15.10! Глядя на код в состоянии покоя, средство проверки типов не может предсказать, что `json.loads()` вернет что-то, напоминающее `BookDict`. Это может гарантировать только проверка во время выполнения.

Статическая проверка типов не может предотвратить ошибки в принципиально динамическом коде, таком как `json.loads()`, который создает объекты Python разных типов во время выполнения, как в примерах 15.11, 15.12 и 15.13.

Пример 15.11. demo_not_book.py: `from_json` возвращает недопустимый `BookDict` и `to_xml` принимает его

```
from books import to_xml, from_json
from typing import TYPE_CHECKING

def demo() -> None:
    NOT_BOOK_JSON = """
        {"title": "Andromeda Strain",
        "flavor": "pistachio",
        "authors": true}
    """
```

[1] Брэтт Кэннон, Гвидо ван Россум и другие обсуждали, как аннотировать `json.loads()`, начиная с 2016 года в проблеме Муру #182 «Define a JSON type» (*https://github.com/python/typing/issues/182*).

```
    not_book = from_json(NOT_BOOK_JSON) ❶
    if TYPE_CHECKING: ❷
        reveal_type(not_book)
        reveal_type(not_book['authors'])

    print(not_book) ❸
    print(not_book['flavor']) ❹

    xml = to_xml(not_book) ❺
    print(xml) ❻

if __name__ == '__main__':
    demo()
```

❶ Эта строка не порождает допустимого `BookDict` – см. состав `NOT_BOOK_JSON`.
❷ Попросим Муру показать два типа.
❸ Здесь не должно быть проблем: `print` способна обработать любой объект и любой тип.
❹ В `BookDict` нет ключа `'flavor'`, но в JSON-коде есть… и что же случится?
❺ Вспомните сигнатуру: `def to_xml(book: BookDict) -> str`:
❻ Как будет выглядеть выходной XML?

Теперь проверим *demo_not_book.py* с помощью Муру (пример 15.12).

Пример 15.12. Отчет Муру для demo_not_book.py, переформатированный для наглядности

```
.../typeddict/ $ mypy demo_not_book.py
demo_not_book.py:12: note: Revealed type is
   'TypedDict('books.BookDict', {'isbn': built-ins.str,
                                  'title': built-ins.str,
                                  'authors': built-ins.list[builtins.str],
                                  'pagecount': built-ins.int})' ❶
demo_not_book.py:13: note: Revealed type is 'built-ins.list[builtins.str]' ❷
demo_not_book.py:16: error: TypedDict "BookDict" has no key 'flavor' ❸
Found 1 error in 1 file (checked 1 source file)
```

❶ Муру показывает номинальный тип, а не состав `not_book` во время выполнения.
❷ И снова это номинальный тип `not_book['authors']`, определенный в `BookDict`. А не тип во время выполнения.
❸ Эта ошибка относится к строке `print(not_book['flavor'])`: такого ключа в номинальном типе нет.

Теперь выполним *demo_not_book.py*, результат показан в примере 15.13.

Пример 15.13. Результат работы `demo_not_book.py`

```
.../typeddict/ $ python3 demo_not_book.py
{'title': 'Andromeda Strain', 'flavor': 'pistachio', 'authors': True} ❶
pistachio ❷
<BOOK>❸
    <TITLE>Andromeda Strain</TITLE>
    <FLAVOR>pistachio</FLAVOR>
    <AUTHORS>True</AUTHORS>
</BOOK>
```

❶ Это не `BookDict`.
❷ Значение `not_book['flavor']`.
❸ `to_xml` принимает аргумент типа `BookDict`, но никакой проверки во время выполнения нет: мусор на входе – мусор на выходе.

Пример 15.13 показывает, что *demo_not_book.py* выводит чепуху, но не выдает ошибок во время выполнения. Использование `TypedDict` при обработке JSON-данных не сильно повысило типобезопасность.

Если взглянуть на код `to_xml` в примере 15.8 сквозь призму утиной типизации, то станет ясно, что аргумент `book` должен предоставлять метод `.items()`, который возвращает итерируемый объект кортежей вида `(key, value)`, где:

- `key` должен иметь метод `.upper()`;
- `value` может быть любым.

Что показала эта демонстрация? При обработке данных с динамической структурой, например в формате JSON или XML, `TypedDict` ни в коей мере не может служить заменой контроля данных во время выполнения. Для этой цели используйте пакет *pydantic* (*https://pypi.org/project/pydantic/*).

У `TypedDict` есть еще много возможностей, в т. ч. поддержка факультативных ключей, ограниченная форма наследования и альтернативный синтаксис объявления. Если хотите узнать больше, поинтересуйтесь документом PEP 589 «TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys» (*https://peps.python.org/pep-0589/*).

Теперь обратимся к функции, которой лучше избегать, но иногда избежать не получается: `typing.cast`.

Приведение типов

Все системы типов несовершенны, включая средства статической проверки типов, аннотации типов в проекте *typeshed* и в сторонних пакетах, в которых они применяются.

Специальная функция `typing.cast()` предлагает способ борьбы с ошибками средств проверки типов или некорректными аннотациями типов, которые мы не можем исправить. Документация по Муру 0.930 (*https://mypy.readthedocs.io/en/stable/type_narrowing.html#casts*) поясняет:

> Приведения используются, чтобы подавить спонтанные предупреждения средства проверки типов и немного помочь ему, когда оно не совсем понимает, что происходит.

На этапе выполнения функция `typing.cast` не делает абсолютно ничего. Вот как выглядит ее реализация:

```
def cast(typ, val):
    """Привести значение к типу.
    Возвращает неизмененное значение. Для средства проверки типов это служит
    сигналом о том, что возвращаемое значение имеет указанный тип, но во время
    выполнения мы намеренно ничего не проверяем (хотим, чтобы программа
    работала максимально быстро).
    """
    return val
```

Документ PEP 484 требует от программ проверки типов «слепой веры» в правильность типа, указанного в `cast`. В разделе «Приведения типов» этого документа (*https://mypy.readthedocs.io/en/stable/type_narrowing.html#casts*) приведен пример, когда средство проверки типов нуждается в помощи со стороны `cast`:

```
from typing import cast

def find_first_str(a: list[object]) -> str:
    index = next(i for i, x in enumerate(a) if isinstance(x, str))
    # Мы попадаем сюда, только если есть хотя бы одна строка
    return cast(str, a[index])
```

Вызов функции `next()` для генераторного выражения либо возвращает индекс элемента типа `str`, либо возбуждает исключение `StopIteration`. Поэтому `find_first_str` всегда возвращает строку, если не было исключения и если в качестве типа возвращаемого значения объявлен `str`.

Но если бы в последней строке мы просто написали `return a[index]`, то Муру вывела бы в качестве возвращаемого типа `object`, потому что аргумент `a` объявлен как `list[object]`. Поэтому руководство со стороны `cast()` необходимо[1].

Приведем еще один пример использования `cast`, на этот раз чтобы исправить устаревшую аннотацию типа в стандартной библиотеке Python. В примере 21.12 я создаю объект *asyncio* `Server` и хочу получить прослушиваемый сервером адрес. Я написал такую строку:

```
addr = server.sockets[0].getsockname()
```

Но Муру отреагировала ошибкой:

```
Value of type «Optional[List[socket]]» is not indexable
```

Аннотация типов для `Server.sockets` в *typeshed* по состоянию на 2021 год корректна для Python 3.6, где атрибут `sockets` мог принимать значение `None`. Но в Python 3.7 `sockets` стало свойством с методом чтения, который всегда возвращает список – возможно, пустой, если сервер не открывал сокетов. А начиная с Python 3.8 метод чтения возвращает кортеж `tuple` (используемый как неизменяемая последовательность).

Поскольку я не могу немедленно исправить *typeshed*[2], я добавил `cast`:

```
from asyncio.trsock import TransportSocket
from typing import cast

# ... много строк опущено ...

    socket_list = cast(tuple[TransportSocket, ...], server.sockets)
    addr = socket_list[0].getsockname()
```

[1] Я намеренно использовал `enumerate` в этом примере, чтобы сбить с толку программу проверки типов. Более простая реализация, которая отдает строки непосредственно, а не пропускает через `enumerate`, была бы правильно проанализирована Муру, и `cast()` не понадобилась бы.

[2] Я зарегистрировал проблему *typeshed* #5535 «Wrong type hint for asyncio.base_events.Server sockets attribute», и ошибку быстро исправил Себастьян Риттау. Однако я решил оставить этот пример, поскольку он иллюстрирует типичную ситуацию, когда `cast` необходима, а мое обращение к `cast` безвредно.

Прежде чем использовать `cast` в этом случае, понадобилось потратить пару часов, чтобы понять проблему и заглянуть в исходный код *asyncio*, дабы найти правильный тип сокетов: класс `TransportSocket` из недокументированного модуля `asyncio.trsock`. Мне также пришлось добавить два предложения `import` и одну строку кода для удобства чтения[1]. Но код стал безопаснее.

Внимательный читатель, возможно, заметил, что вычисление `sockets[0]` может возбудить исключение `IndexError`, если список `sockets` пуст. Однако, насколько я понимаю `asyncio`, в примере 21.12 такого случиться не может, потом что `server` готов принимать запросы на подключение к тому моменту, как я прочитал атрибут `sockets`, поэтому пустым список не будет. Как бы то ни было, `IndexError` – ошибка времени выполнения, поэтому Муру не может обнаружить ее даже в тривиальном случае, таком как `print([][0])`.

Не стоит чувствовать себя комфортно, заткнув Муру рот с помощью `cast`, потому что Муру обычно права, когда сообщает об ошибке. Слишком частое использование `cast` дурно пахнет. Быть может, ваша команда неправильно пользуется аннотациями типов или в вашей кодовой базе имеются зависимости низкого качества.

Несмотря на все недостатки, существуют ситуации, когда использование `cast` оправдано. Вот что писал об этом Гвидо ван Россум:

> Что дурного, если вы невзначай вызовете `cast()` или напишете комментарий `# type: ignore`[2]?

Немудро полностью отказываться от `cast`, особенно когда другие обходные пути еще хуже:

- `# type: ignore` менее информативно[3];
- использование `Any` заразно: поскольку `Any` *совместим со* всеми типами, злоупотребление им может вызвать каскадный эффект вследствие вывода типов, что сводит на нет попытки программы проверки типов обнаружить ошибки в других частях кода.

Конечно, не все проблемы типизации можно исправить с помощью `cast`. Иногда необходим комментарий `# type: ignore`, иногда тип `Any`, а иногда даже лучше оставить функцию без аннотаций типов.

Далее мы поговорим об использовании аннотаций на этапе выполнения.

---

[1] Честно говоря, я сначала добавил комментарий `# type: ignore` в строку с `server.sockets[0]`, потому что после недолгих изысканий нашел похожие строки в документации по *asyncio* и в тесте, поэтому заподозрил, что проблема не в моем коде.

[2] Сообщение от 19 мая 2020 года (*https://mail.python.org/archives/list/typing-sig@python.org/message/5LCWMN2UY2UQNLC5Z47GHBZKSPZW4I63/*) в списке рассылки typing-sig.

[3] Синтаксическая конструкция `# type: ignore[code]` позволяет указать, какой код ошибки Муру следует подавить, но коды не всегда легко интерпретировать. См. раздел «Коды ошибок» (*https://mypy.readthedocs.io/en/stable/error_codes.html#error-codes*) в документации по Муру.

Чтение аннотаций типов во время выполнения

На этапе импорта Python читает аннотации типов в функциях, классах и модулях и сохраняет их в атрибутах `__annotations__`. Рассмотрим функцию `clip` в примере 15.14[1].

Пример 15.14. clipannot.py: аннотированная сигнатура функции `clip`

```
def clip(text: str, max_len: int = 80) -> str:
```

Аннотации типов хранятся в атрибуте функции `__annotations__`, представляющем собой словарь:

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': <class 'int'>, 'return':
<class 'str'>}
```

Ключу `'return'` соответствует аннотация возвращаемого типа после символа `->`.

Заметим, что аннотации обрабатываются интерпретатором на этапе импорта, тогда же, когда значения по умолчанию. Именно поэтому значениями в аннотациях являются классы Python `str` и `int`, а не строки `'str'` и `'int'`. Вычисление аннотаций во время импорта является стандартом в версии Python 3.10, но это положение дел может измениться, если стандартным станет поведение, описанное в документах PEP 563 (*https://peps.python.org/pep-0563/*) или PEP 649 (*https://peps.python.org/pep-0649/*).

Проблемы с аннотациями во время выполнения

Расширяющееся использование аннотаций типов подняло две проблемы:

- импорт модулей занимает больше времени и потребляет больше памяти, если используется много аннотаций;
- ссылка на еще не определенные типы требует использования строк, а не фактических типов.

Обе проблемы релевантны. Первая – потому что мы только что видели: аннотации вычисляются интерпретатором на этапе импорта и сохраняются в атрибуте `__annotations__`. Сосредоточимся на второй.

Хранение аннотаций в виде строк иногда необходимо из-за проблемы «опережающей ссылки»: когда аннотация типа должна сослаться на класс, определенный в том же модуле ниже. Однако типичное проявление этой проблемы в исходном коде вовсе не наводит на мысль об опережающей ссылке: просто метод возвращает новый объект того же класса. Поскольку объект класса не определен, пока Python не закончит вычисление тела класса, в аннотациях типов необходимо указывать имя класса в виде строки. Приведем пример:

```
class Rectangle:
    # ... строки опущены ...
    def stretch(self, factor: float) -> 'Rectangle':
        return Rectangle(width=self.width * factor)
```

[1] Я не стану углубляться в реализацию `clip`, но если вам интересно, можете прочитать код всего модуля в файле *clip_annot.py* (*https://github.com/fluentpython/example-code-2e/blob/master/15-more-types/clip_annot.py*).

Запись еще не определенных типов в виде строки в аннотациях типов – стандартная и обязательная практика в версии Python 3.10. Средства статической проверки типов с самого начала проектировались с учетом этой проблемы.

Но если вы напишете код, который во время выполнения читает аннотацию `return` для функции `stretch`, то получите строку `'Rectangle'`, а не ссылку на фактический тип, класс `Rectangle`. Теперь ваша программа должна понять, что эта строка означает.

В модуле `typing` есть три функции и класс, отнесенные к категории помощников интроспекции (*https://docs.python.org/3/library/typing.html#introspection-helpers*). Самая важная из них `typing.get_type_hints`. В документации читаем:

`get_type_hints(obj, globals=None, locals=None, include_extras=False)`

> [...] Часто это то же самое, что `obj.__annotations__`. Кроме того, опережающие ссылки, представленные строковыми литералами, обрабатываются путем вычисления их в пространствах имен `globals` и `locals`. [...]

Начиная с Python 3.10 следует использовать новую функцию `inspect.get_annotations(...)`, а не `typing.get_type_hints`. Однако у некоторых читателей Python 3.10, возможно, еще не установлена, поэтому в примерах ниже я пользуюсь `typing. get_ type_ hints`, которая доступна с момента появления модуля `typing` в Python 3.5.

Одобренный документ PEP 563 «Postponed Evaluation of Annotations» (*https://peps.python.org/pep-0563/*) сделал необязательной запись аннотаций в виде строк и уменьшил время, необходимое для обработки аннотаций типов во время выполнения. Его основная идея описана в следующих двух предложениях в разделе «Реферат»:

> В этом PEP предлагается изменить аннотации функций и переменных, так чтобы они больше не вычислялись в момент определения функции. Вместо этого они сохраняются в аннотациях в строковой форме.

Начиная с версии Python 3.7 именно так обрабатываются аннотации в любом модуле, который начинается следующим предложением `import`:

```
from __future__ import annotations
```

Для демонстрации последствий я поместил копию функции `clip` из примера 15.14 в модуль *clip_annot_post.py*, в начале которого находится этот импорт из `__future__`.

Вот что было напечатано на консоли, когда я импортировал этот модуль и прочитал аннотации из `clip`:

```
>>> from clip_annot_post import clip
>>> clip.__annotations__
{'text': 'str', 'max_len': 'int', 'return': 'str'}
```

Как видите, теперь все аннотации типов стали простыми строками, хотя в определении `clip` и не были записаны как строки в кавычках (пример 15.14).

Функция `typing.get_type_hints` умеет разрешать многие аннотации типов, включая и те, что есть в `clip`:

```
>>> from clip_annot_post import clip
>>> from typing import get_type_hints
>>> get_type_hints(clip)
{'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

Вызов `get_type_hints` дает нам реальные типы – даже в тех случаях, когда в исходной аннотации тип был записан в виде закавыченной строки. Это рекомендуемый способ читать аннотации типов во время выполнения.

Поведение, описанное в документе PEP 563, предполагалось сделать подразумеваемым по умолчанию в версии Python 3.10 и тем самым избавиться от необходимости импортировать `__future__`. Но ответственные за сопровождение *FastAPI* и *pydantic* подняли тревогу, заявив, что это изменение поломает их код, который зависит от чтения аннотаций типов во время выполнения и не может надежно использовать функцию `get_type_hints`.

В последующем обсуждении в списке рассылки python-dev Лукаш Ланга, автор PEP 563, описал некоторые ограничения этой функции:

> [...] оказалось, что у функции `typing.get_type_hints()` есть ограничения, из-за которых ее использование во время выполнения вообще обходится дорого, но, что важнее, ее недостаточно для разрешения всех типов. Самый типичный пример имеет отношение к неглобальному контексту, в котором генерируются типы (например, внутренние классы, классы внутри функций и т. д.). Но один из коронных примеров опережающих ссылок – классы с методами, которые принимают или возвращают объекты их собственного типа, также некорректно обрабатываются функцией `typing.get_type_hints()`, если используется генератор классов. Можно пойти на кое-какие трюки, чтобы связать всё воедино, но в общем случае это не есть хорошо[1].

Руководящий комитет Python решил отложить внедрение документа PEP 563 до версии Python 3.11 или более поздней, чтобы дать разработчикам больше времени на поиск решения тех проблем, которые пытался решить PEP 563, не ставя под угрозу широко распространенное применение аннотаций типов во время выполнения. Документ PEP 649 «Deferred Evaluation Of Annotations Using Descriptors» (*https://peps.python.org/pep-0649/*) обсуждается как возможное решение, но, возможно, будет достигнут другой компромисс.

Итак: чтение аннотаций типов не стопроцентно надежно в версии Python 3.10, и это положение вряд ли изменится в 2022 году.

Компании, в которых Python используется для разработки крупномасштабных проектов, нуждаются в преимуществах статической типизации, но не хотят платить высокую цену за вычисление аннотаций типов на этапе импорта. Статическая проверка производится на машинах разработчиков и на выделенных серверах непрерывной интеграции, но загрузка модулей выполняется куда чаще в производственных контейнерах, и пренебречь затратами на нее в крупном проекте никак нельзя.

В сообществе Python это является причиной трений между теми, кто желает хранить аннотации типов только в виде строк – чтобы уменьшить накладные расходы на этапе загрузки, – и теми, кто

[1] Сообщение «PEP 563 in light of PEP 649» (*https://mail.python.org/archives/list/python-dev@python.org/message/ZBI7MD6CSGM6LZAOTET7GXAVBZB7O77O/*), отправлено 16 апреля 2021.

хочет использовать их также во время выполнения, к каковым, в частности, относятся авторы и пользователи пакетов *pydantic* и *FastAPI*, которые предпочли бы иметь под рукой готовые объекты типов, а не вычислять аннотации, что довольно накладно.

Как решать проблему

Учитывая нестабильность ситуации, сложившейся на настоящий момент, я могу дать следующие рекомендации тем, кому нужно читать аннотации во время выполнения.

- Избегайте чтения `__annotations__` напрямую; вместо этого пользуйтесь функциями `inspect.get_annotations` (начиная с Python 3.10) или `typing.get_type_hints` (начиная с Python 3.5).
- Напишите свою функцию, которая будет служить тонкой оберткой вокруг `inspect.get_annotations` или `typing.get_type_hints`, а в остальной части своей кодовой базы вызывайте эту функцию, тогда будущие изменения будут локализованы только в ней.

Для демонстрации второго подхода ниже приведены начальные строки класса `Checked`, определенного в примере 24.5, который мы подробно изучим в главе 24:

```
class Checked:
    @classmethod
    def _fields(cls) -> dict[str, type]:
        return get_type_hints(cls)
    # ... еще строки ...
```

Метод класса `Checked._fields` экранирует другие части модуля от прямой зависимости от `typing.get_type_hints`. Если в будущем `get_type_hints` изменится, так что понадобится дополнительная логика, или мы захотим заменить ее функцией `inspect.get_annotations`, то изменение будет ограничено методом `Checked._fields` и не повлияет на остальную программу.

С учетом продолжающихся дискуссий и предлагаемых изменений в порядке инспекции аннотаций типов во время выполнения официальный документ «Annotations Best Practices» (*https://docs.python.org/3.10/howto/annotations.html*) следует прочитать обязательно, и очень может статься, что по пути к Python 3.11 он будет обновлен. Эта инструкция написана Ларри Хастингсом, автором документа PEP 649 «Deferred Evaluation Of Annotations Using Descriptors» (*https://peps.python.org/pep-0649/*), альтернативного предложения для решения проблем, поднятых в документе PEP 563 «Postponed Evaluation of Annotations» (*https://peps.python.org/pep-0563/*).

Далее в этой главе мы рассмотрим обобщенные типы и начнем с того, как определить обобщенный класс, допускающий параметризацию пользователями.

Реализация обобщенного класса

В примере 13.7 мы определили ABC `Tombola`: интерфейс для классов, работающих как лотерейный барабан. Класс `LottoBlower` из примера 13.10 является кон-

кретной реализацией. Теперь мы рассмотрим обобщенную версию `LottoBlower`, использование которой показано в примере 15.15.

Пример 15.15. generic_lotto_demo.py: использование обобщенного класса для розыгрыша лотереи

```
from generic_lotto import LottoBlower

machine = LottoBlower[int](range(1, 11)) ❶

first = machine.pick() ❷
remain = machine.inspect() ❸
```

❶ Для создания экземпляра обобщенного класса мы передаем ему фактический параметр-тип, в данном случае `int`.
❷ Муру правильно выводит, что `first` имеет тип `int`...
❸ ... и что `remain` – кортеж `tuple` целых чисел.

Дополнительно Муру сообщает о нарушениях при использовании параметризованного типа, выводя полезные сообщения, показанные в примере 15.16.

Пример 15.16. generic_lotto_errors.py: сообщения об ошибках, выданные Муру

```
from generic_lotto import LottoBlower

machine = LottoBlower[int]([1, .2])
## error: List item 1 has incompatible type "float"; ❶
##        expected "int"

machine = LottoBlower[int](range(1, 11))

machine.load('ABC')
## error: Argument 1 to "load" of "LottoBlower" ❷
##        has incompatible type "str";
##        expected "Iterable[int]"
## note: Following member(s) of "str" have conflicts:
## note: Expected:
## note: def __iter__(self) -> Iterator[int]
## note: Got:
## note: def __iter__(self) -> Iterator[str]
```

❶ После создания `LottoBlower[int]` Муру помечает использование `float` как ошибку.
❷ При вызове `.load('ABC')` Муру объясняет, почему `str` не годится: `str.__iter__` возвращает `Iterator[str]`, но `LottoBlower[int]` требует `Iterator[int]`.

В примере 15.17 приведена реализация.

Пример 15.17. generic_lotto.py: обобщенный класс для розыгрыша лотереи

```
import random

from collections.abc import Iterable
from typing import TypeVar, Generic

from tombola import Tombola

T = TypeVar('T')
```

```
class LottoBlower(Tombola, Generic[T]): ❶

    def __init__(self, items: Iterable[T]) -> None: ❷
        self._balls = list[T](items)

    def load(self, items: Iterable[T]) -> None: ❸
        self._balls.extend(items)

    def pick(self) -> T: ❹
        try:
            position = random.randrange(len(self._balls))
        except ValueError:
            raise LookupError('pick from empty LottoBlower')
        return self._balls.pop(position)

    def loaded(self) -> bool: ❺
        return bool(self._balls)

    def inspect(self) -> tuple[T, ...]: ❻
        return tuple(self._balls)
```

❶ В объявлениях обобщенных классов часто используется множественное наследование, поскольку мы должны унаследовать `Generic`, чтобы объявить формальные параметры-типы – в данном случае `T`.
❷ Аргумент `items` метода `__init__` имеет тип `Iterable[T]`, который превращается в `Iterable[int]`, когда экземпляр объявляется как `LottoBlower[int]`.
❸ Метод `load` имеет аналогичные ограничения.
❹ Типом возвращаемого значения `T` в `LottoBlower[int]` становится `int`.
❺ Здесь никакой переменной-типа нет.
❻ Наконец, `T` определяет тип элементов в возвращенном кортеже.

В коротком разделе «Пользовательские обобщенные типы» (*https://docs.python.org/3/library/typing.html#user-defined-generic-types*) документации по модулю `typing` приведены хорошие примеры и представлены дополнительные детали, которых я здесь не рассматриваю.

Итак, мы посмотрели, как реализуется обобщенный класс, а теперь определим терминологию, употребляемую при обсуждении обобщенных типов.

Основы терминологии, относящейся к обобщенным типам

Ниже приведено несколько определений, которые, на мой взгляд, полезны при изучении обобщенных типов[1].

Обобщенный тип

Тип, в объявлении которого присутствует одна или несколько переменных-типов.

Примеры: `LottoBlower[T]`, `abc.Mapping[KT, VT]`.

[1] Термины взяты из классической книги Joshua Bloch «Effective Java», 3-е издание (Addison-Wesley). Определения и примеры мои.

Формальный параметр-тип

Переменные-типы, встречающиеся в объявлении обобщенного типа.
Пример: `KT` и `VT` в предыдущем примере `abc.Mapping[KT,VT]`.

Параметризованный тип

Тип, объявленный с фактическими параметрами-типами.
Примеры: `LottoBlower[int]`, `abc.Mapping[str, float]`.

Фактический параметр-тип

Фактические типы, заданные в качестве параметров в объявлении параметризованного типа.
Пример: `int` в объявлении `LottoBlower[int]`.

В следующем разделе мы обсудим, как сделать обобщенные типы более гибкими, и введем понятия ковариантности, контравариантности и инвариантности.

Вариантность

В зависимости от вашего опыта работы с обобщенными типами в других языках этот раздел может оказаться самым трудным во всей книге. Концепция вариантности абстрактна, и попытка изложить ее строго сделала бы эту книгу похожей на учебник математики.

На практике вариантность больше интересна авторам библиотек, которые хотят поддержать новые обобщенные контейнерные типы или предоставить API на основе обратных вызовов. Но даже тогда излишней сложности можно избежать, ограничившись только поддержкой инвариантных контейнеров, а именно таково большинство контейнеров в стандартной библиотеке Python. Так что при первом чтении можете пропустить этот раздел целиком или прочитать только те его части, которые относятся к инвариантным типам.

Впервые с понятием *вариантности* мы столкнулись в разделе «Вариантность в типах Callable» главы 8 применительно к параметризованным обобщенным типам `Callable`. Здесь же мы распространим эту концепцию на обобщенные типы коллекций, пользуясь аналогией с «реальным миром», чтобы наполнить абстрактную концепцию конкретикой.

Допустим, что в школьной столовой действует правило: разрешено устанавливать только автоматы для розлива соков[1]. Разливать любые напитки не разрешается, чтобы не поить детей газировкой, которая запрещена советом школы[2].

Инвариантный разливочный автомат

Попробуем смоделировать описанную ситуацию с помощью обобщенного класса `BeverageDispenser`, который можно параметризовать типом напитка.

[1] Впервые я встретил аналогию со столовой при обсуждении вариантности в предисловии Эрика Мейера к книге Gilad Bracha «The Dart Programming Language» (Addison-Wesley).

[2] Гораздо лучше, чем запрещать книги!

Пример 15.18. invariant.py: определения типов и функция `install`

```
from typing import TypeVar, Generic

class Beverage: ❶
    """Любой напиток."""

class Juice(Beverage):
    """Любой фруктовый сок."""

class OrangeJuice(Juice):
    """Восхитительный сок бразильских апельсинов."""

T = TypeVar('T') ❷

class BeverageDispenser(Generic[T]): ❸
    """Автомат, параметризованный типом напитка."""
    def __init__(self, beverage: T) -> None:
        self.beverage = beverage

    def dispense(self) -> T:
        return self.beverage

    def install(dispenser: BeverageDispenser[Juice]) -> None: ❹
        """Установить автомат для розлива фруктовых соков."""
```

❶ `Beverage`, `Juice` и `OrangeJuice` из иерархии типов.
❷ Простое объявление `TypeVar`.
❸ `BeverageDispenser` параметризован типом напитка.
❹ Функция `install` глобальна на уровне модуля. В ее аннотации типа указано, что допустим только автомат для розлива соков.

При определениях в примере 15.18 следующий код допустим:

```
juice_dispenser = BeverageDispenser(Juice())
install(juice_dispenser)
```

Но показанный ниже код недопустим:

```
beverage_dispenser = BeverageDispenser(Beverage())
install(beverage_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[Beverage]"
##       expected "BeverageDispenser[Juice]"
```

Автомат, разливающий любой напиток `Beverage`, недопустим, потому что согласно правилам столовой разрешены только автоматы для розлива соков `Juice`.

Удивительно, но этот код тоже недопустим:

```
orange_juice_dispenser = BeverageDispenser(OrangeJuice())
install(orange_juice_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[OrangeJuice]"
##       expected "BeverageDispenser[Juice]"
```

Автомат, специализированный для розлива апельсинового сока `OrangeJuice`, тоже недопустим. Подходит только `BeverageDispenser[Juice]`. Говорят, что тип

`BeverageDispenser(Generic[T])`, если `BeverageDispenser[OrangeJuice]` не совместим с `BeverageDispenser[Juice]`, несмотря на то что `OrangeJuice` *является подтипом* `Juice`.

Типы изменяемых коллекций в Python, например `list` и `set`, инвариантны. Класс `LottoBlower` из примера 15.17 тоже инвариантен.

Ковариантный разливочный автомат

Если мы хотим большей гибкости, т. е. нужно моделировать разливочные автоматы как обобщенный класс, который принимает тип напитка и его подтипы, то нужно сделать этот класс ковариантным. В примере 15.19 показано, как следует объявить `BeverageDispenser`.

Пример 15.19. covariant.py: определения типов и функция `install`

```
T_co = TypeVar('T_co', covariant=True) ❶

class BeverageDispenser(Generic[T_co]): ❷
    def __init__(self, beverage: T_co) -> None:
        self.beverage = beverage

    def dispense(self) -> T_co:
        return self.beverage

def install(dispenser: BeverageDispenser[Juice]) -> None: ❸
    """Установить автомат для розлива фруктовых соков."""
```

❶ Установить `covariant=True` при объявлении переменной-типа; по соглашению суффикс `_co` в *typeshed* обозначает ковариантные параметры-типы.
❷ Использовать `T_co` для параметризации специального класса `Generic`.
❸ Аннотации типов для `install` такие же, как в примере 15.18.

Следующий код работает, потому что теперь ковариантный тип `BeverageDispenser` принимает как тип `Juice`, так и тип `OrangeJuice`:

```
juice_dispenser = BeverageDispenser(Juice())
install(juice_dispenser)

orange_juice_dispenser = BeverageDispenser(OrangeJuice())
install(orange_juice_dispenser)
```

Но автомат для розлива произвольных напитков `Beverage` не принимается:

```
beverage_dispenser = BeverageDispenser(Beverage())
install(beverage_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[Beverage]"
##       expected "BeverageDispenser[Juice]"
```

Это была ковариантность: связь тип–подтип между параметризованными автоматами изменяется в том же направлении, что и связь тип–подтип между параметрами-типами.

Контравариантная урна

Теперь смоделируем правило по установке урн для мусора. Предположим, что еда и напитки поставляются в биоразлагаемых упаковках и пищевые отходы

и одноразовые столовые приборы тоже биоразлагаемые. Урны должны быть пригодны для биоразлагаемых отходов.

В педагогических целях сделаем упрощающее предположение о том, что мусор можно организовать в виде простой иерархии:

- `Refuse` – самый общий тип отходов;
- `Biodegradable` – специальный тип отходов, который со временем разлагается микроорганизмами. Некоторые виды отходов не являются биоразлагаемыми;
- `Compostable` – специальный тип биоразлагаемых отходов, который можно эффективно превратить в органическое удобрение в компостном баке или в установке компостирования. Не всякие биоразлагаемые отходы являются компостируемыми в смысле нашего определения.

Чтобы смоделировать правило установки урн в столовой, нам придется ввести концепцию контравариантности. Мы сделаем это на примере.

Пример 15.20. contravariant.py: определения типов и функция `install`

```
from typing import TypeVar, Generic

class Refuse: ❶
    """Любые отходы."""

class Biodegradable(Refuse):
    """Биоразлагаемые отходы."""

class Compostable(Biodegradable):
    """Компостируемые отходы."""

    T_contra = TypeVar('T_contra', contravariant=True) ❷

class TrashCan(Generic[T_contra]): ❸
    def put(self, refuse: T_contra) -> None:
        """Хранить отходы, пока не выгружены."""

    def deploy(trash_can: TrashCan[Biodegradable]):
        """Установить урну для биоразлагаемых отходов."""
```

❶ Иерархия типов для отходов: `Refuse` – самый общий тип, `Compostable` – самый специфичный.
❷ `T_contra` – принятое по соглашению имя контравариантной переменной-типа.
❸ `TrashCan` контравариантен относительно типа отходов.

При таких определениях следующие типы урн допустимы:

```
bio_can: TrashCan[Biodegradable] = TrashCan()
deploy(bio_can)

trash_can: TrashCan[Refuse] = TrashCan()
deploy(trash_can)
```

Более общий тип `TrashCan[Refuse]` допустим, потому что может содержать любые типы отходов, в т. ч. `Biodegradable`. Но `TrashCan[Compostable]` не годится, потому что не может содержать `Biodegradable`:

```
compost_can: TrashCan[Compostable] = TrashCan()
deploy(compost_can)
## mypy: Argument 1 to "deploy" has incompatible type "TrashCan[Compostable]"
##       expected "TrashCan[Biodegradable]"
```

Подведем итоги.

Обзор вариантности

Вариантность – непростое свойство. В следующих подразделах мы повторим, что такое инвариантные, ковариантные и контравариантные типы, и предложим несколько эвристических правил, позволяющих рассуждать о них.

Инвариантные типы

Обобщенный тип `L` инвариантен, если между двумя параметризованными типами нет отношения тип–подтип, даже если такое отношение существует между фактическими параметрами. Иными словами, если `L` инвариантен, то `L[A]` не является ни подтипом, ни супертипом `L[B]`. Они несовместимы в обоих направлениях.

Как уже отмечалось, изменяемые коллекции в Python по умолчанию инвариантны. Хорошим примером может служить тип `list`: `list[int]` не *совместим с* `list[float]`, и наоборот.

В общем случае, если формальный параметр-тип встречается в аннотациях типов аргументов метода и тот же параметр встречается в типе возвращаемого методом значения, то параметр должен быть инвариантен, чтобы гарантировать типобезопасность при обновлении коллекции и чтении из нее.

Например, ниже приведена часть аннотаций типов для встроенного типа `list` в *typeshed*:

```
class list(MutableSequence[_T], Generic[_T]):
    @overload
    def __init__(self) -> None: ...
    @overload
    def __init__(self, iterable: Iterable[_T]) -> None: ...
    # ... строки опущены...
    def append(self, __object: _T) -> None: ...
    def extend(self, __iterable: Iterable[_T]) -> None: ...
    def pop(self, __index: int = ...) -> _T: ...
    # и т. д. ...
```

Заметим, что `_T` встречается в аргументах `__init__`, `append` и `extend`, а также как тип значения, возвращаемого `pop`. Невозможно сделать такой класс типобезопасным, если он ковариантен или контравариантен относительно `_T`.

Ковариантные типы

Рассмотрим два типа `A` и `B`, где `B` *совместим с* `A` и ни один из них не совпадает с `Any`. Некоторые авторы используют символы `<:` и `:>`, чтобы обозначить следующие отношения:

`A :> B`
: `A` является *супертипом* или совпадает с `B`.

`B <: A`
: `B` является *супертипом* или совпадает с `A`.

Если `A :> B`, тот обобщенный тип `C` ковариантен, когда `C[A] :> C[B]`.

Обратите внимание, что направление символа `:>` одинаково в обоих случаях, когда `A` встречается слева от `B`. Ковариантные обобщенные типы повторяют отношение тип–подтип между фактическими параметрами-типами.

Неизменяемые контейнеры могут быть ковариантными. Например, в документации (*https://docs.python.org/3.10/library/typing.html#typing.FrozenSet*) написано, что класс `typing.FrozenSet` ковариантен относительно переменной-типа, и для выражения этой мысли в соответствии с принятым соглашением используется имя `T_co`:

```
class FrozenSet(frozenset, AbstractSet[T_co]):
```

Применяя нотацию `:>` к параметризованным типам, имеем:

```
           float :> int
frozenset[float] :> frozenset[int]
```

Итераторы дают еще один пример ковариантных обобщенных типов: они не являются коллекциями, допускающими только чтение, как `frozenset`, а лишь порождают выход. Любой код, ожидающий итератор `abc.Iterator[float]`, который отдает числа с плавающей точкой, может безопасно использовать итератор `abc.Iterator[int]`, отдающие целые. По той же причине типы `Callable` ковариантны относительно типа возвращаемого значения.

Контравариантные типы

Если `A :> B`, то обобщенный тип `K` контравариантен, если `if K[A] <: K[B]`.

Контравариантные обобщенные типы обращают связь тип–подтип между фактическими параметрами-типами.

Примером может служить класс `TrashCan`:

```
          Refuse :> Biodegradable
TrashCan[Refuse] <: TrashCan[Biodegradable]
```

Контравариантный контейнер обычно представляет собой структуру данных, предназначенную только для записи и называемую «стоком». В стандартной библиотеке нет таких коллекций, но есть несколько типов с контравариантными параметрами-типами.

Тип `Callable[[ParamType, ...], ReturnType]` контравариантен относительно параметров-типов, но ковариантен относительно `ReturnType`, как мы видели в разделе «Вариантность в типах Callable» главы 8. Кроме того, типы `Generator`, `Coroutine` и `AsyncGenerator` имеют по одному контравариантному параметру-типу. Тип `Generator` описан в разделе «Обобщенные аннотации типов для классических сопрограмм» главы 17, а типы `Coroutine` и `AsyncGenerator` обсуждаются в главе 21.

В этом обсуждении вариантности главное то, что контравариантные формальные параметры определяют типы аргументов, используемых для вызова или отправки данных объекту, тогда как ковариантные формальные параметры определяют типы выходов, порождаемых объектом, – тип отдаваемого или возвращаемого значения, в зависимости от объекта. Семантика терминов «отправить» и «отдать» объясняется в разделе «Классические сопрограммы» главы 17.

Из этих наблюдений над ковариантными выходами и контравариантными входами можно вывести ряд полезных рекомендаций.

Эвристические правила вариантности

Напоследок приведем несколько эвристических правил, полезных при рассуждениях о вариантности.

- Если формальный параметр-тип определяет тип данных, исходящих из объекта, то он может быть ковариантным.
- Если формальный параметр-тип определяет тип данных, входящих в объект после его начального конструирования, то он может быть контравариантным.
- Если формальный параметр-тип определяет тип данных, исходящих из объекта, и тот же параметр определяет тип данных, входящих в объект, то он должен быть инвариантным.
- Чтобы ненароком не допустить ошибку, делайте формальные параметры инвариантными.

Тип `Callable[[ParamType, ...], ReturnType]` служит иллюстрацией правил 1 и 2: `ReturnType` ковариантный, а каждый `ParamType` контравариантный.

По умолчанию `TypeVar` создает инвариантные формальные параметры, и именно так аннотированы изменяемые коллекции в стандартной библиотеке. В разделе «Обобщенные аннотации типов для классических сопрограмм» главы 17 мы продолжим обсуждение вариантности.

А теперь посмотрим, как определить обобщенные статические протоколы, применив идею ковариантности к двум новым примерам.

Реализация обобщенного статического протокола

В стандартной библиотеке Python 3.10 есть несколько обобщенных статических протоколов. Один из них, `SupportsAbs`, реализован в модуле *typing* (*https://github.com/python/cpython/blob/46b16d0bdbb1722daed10389e27226a2370f1635/Lib/typing.py#L1786)*, как показано ниже.

```
@runtime_checkable
class SupportsAbs(Protocol[T_co]):
    """ABC с одним абстрактным методом __abs__, ковариантным относительно типа
       возвращаемого значения."""
    __slots__ = ()

    @abstractmethod
    def __abs__(self) -> T_co:
        pass
```

`T_co` объявлен в соответствии с соглашением об именовании:

```
T_co = TypeVar('T_co', covariant=True)
```

Благодаря `SupportsAbs` Муру считает код в примере 15.21 допустимым.

Пример 15.21. abs_demo.py: использование обобщенного протокола `SupportsAbs`

```
import math
from typing import NamedTuple, SupportsAbs

class Vector2d(NamedTuple):
    x: float
```

```
    y: float

    def __abs__(self) -> float: ❶
        return math.hypot(self.x, self.y)

def is_unit(v: SupportsAbs[float]) -> bool: ❷
    """'True', если абсолютная величина 'v' близка к 1."""
    return math.isclose(abs(v), 1.0) ❸

assert issubclass(Vector2d, SupportsAbs) ❹

v0 = Vector2d(0, 1) ❺
sqrt2 = math.sqrt(2)
v1 = Vector2d(sqrt2 / 2, sqrt2 / 2)
v2 = Vector2d(1, 1)
v3 = complex(.5, math.sqrt(3) / 2)
v4 = 1 ❻

assert is_unit(v0)
assert is_unit(v1)
assert not is_unit(v2)
assert is_unit(v3)
assert is_unit(v4)

print('OK')
```

❶ Определение `__abs__` делает `Vector2d` *совместимым с* `SupportsAbs`.
❷ Параметризация `SupportsAbs` типом `float` гарантирует, что ...
❸ ...Муру будет считать `abs(v)` допустимым первым аргументом `math.isclose`.
❹ Благодаря декоратору `@runtime_checkable` в определении `SupportsAbs` это допустимое утверждение времени выполнения.
❺ Весь остальной код проходит проверки Муру и утверждения времени выполнения.
❻ Тип `int` также *совместим с* `SupportsAbs`. Согласно *typeshed* `int.__abs__` возвращает `int`, который *совместим с* параметром-типом `float`, объявленным в аннотации типа для аргумента `v` функции `is_unit`.

Аналогично можно написать обобщенную версию протокола `RandomPicker`, представленного в примере 13.18, в котором был определен единственный метод `pick`, возвращающий `Any`.

В примере 15.22 показано, как сделать обобщенный `RandomPicker` ковариантным относительно типа, возвращаемого `pick`.

Пример 15.22. generic_randompick.py: определение обобщенного `RandomPicker`

```
from typing import Protocol, runtime_checkable, TypeVar

T_co = TypeVar('T_co', covariant=True) ❶

@runtime_checkable
class RandomPicker(Protocol[T_co]): ❷
    def pick(self) -> T_co: ... ❸
```

❶ Объявить `T_co` ковариантным.

❷ Это делает `RandomPicker` обобщенным типом с ковариантным формальным параметром-типом.
❸ Использовать `T_co` в качестве типа возвращаемого значения.

Обобщенный протокол `RandomPicker` может быть ковариантным, потому что его единственный формальный параметр встречается в типе возвращаемого значения.

На этом мы можем с чистой совестью закончить главу.

Резюме

Мы начали эту главу с простого примера использования декоратора `@overload`, после чего привели и подробно изучили гораздо более сложный пример: как правильно аннотировать встроенную функцию `max` с помощью перегруженных сигнатур.

Далее мы перешли к специальной конструкции `typing.TypedDict`. Я решил рассмотреть ее здесь, а не в главе 5, где шла речь о классе `typing.NamedTuple`, потому что `TypedDict` – не построитель класса, а просто способ добавить аннотации типов к переменной или аргументу, которые нуждаются в словаре, содержащем конкретный набор строковых ключей и определенных типов для каждого ключа – так бывает, когда мы используем `dict` как запись, часто в контексте обработки JSON-данных. Этот раздел получился довольно длинным, потому что использование `TypedDict` иногда вселяет ложное ощущение безопасности, а я хотел показать, что от проверок во время выполнения и обработки ошибок не уйти, если мы пытаемся построить статически структурированные записи из динамических по своей природе отображений.

Затем мы поговорили о функции `typing.cast`, позволяющей руководить работой средства проверки типов. Важно хорошенько продумывать, когда стоит использовать `cast`, поскольку злоупотребление ей может свести на нет все преимущества программы проверки типов.

Доступ к аннотациям типов во время выполнения стал следующей нашей темой. Здесь главный посыл заключался в том, что нужно использовать функцию `typing.get_type_hints`, а не читать атрибут `__annotations__` напрямую. Однако для некоторых аннотаций эта функция может работать ненадежно, и мы видели, что разработчики ядра Python все еще трудятся над тем, как сделать аннотации типов доступными во время выполнения, уменьшив при этом потребление процессора и памяти.

Последние разделы были посвящены обобщенным типам. Мы начали с обобщенного класса `LottoBlower`, который, как мы впоследствии узнали, является инвариантным. За этим примером последовали определения четырех основных терминов: обобщенный тип, формальный параметр-тип, параметризованный тип и фактический параметр-тип.

Далее был рассмотрен важный вопрос о вариантности на примере автоматов для розлива напитков и мусорных урн, взятых из «реальной жизни» и демонстрирующих инвариантные, ковариантные и контравариантные обобщенные типы. Затем мы формализовали и применили эти понятия к примерам из стандартной библиотеки Python.

Напоследок показали, как определяется обобщенный статический протокол, для чего сначала рассмотрели протокол `typing.SupportsAbs`, а затем применили ту же идею к типу `RandomPicker`, сделав его более строгим, чем оригинальный протокол из главы 13.

Система типов в Python – обширная тема, которая к тому же быстро развивается. Эта глава не претендует на полноту. Я выбрал те аспекты, которые либо нашли широкое применение, либо особенно трудны, либо концептуально важны и потому, вероятно, будут оставаться актуальными еще долго.

Дополнительная литература

Статическая система типов в Python была сложна уже на этапе начального проектирования и с каждым годом становилась только сложнее. В табл. 15.1 перечислены все известные мне документы PEP по состоянию на май 2021 года. Понадобилась бы целая книга, чтобы рассмотреть каждый из них.

Таблица 15.1. Документы PEP, посвященные аннотациям типов со ссылками. Звездочкой отмечены PEP, достаточно важные, чтобы заслужить упоминание во вступительном абзаце документации по модулю typing. Вопросительный знак в столбце Python означает, что PEP еще обсуждается или пока не реализован; – означает, что PEP информационный и не связан ни с какой конкретной версией Python

PEP	Название	Python	Год
3107	Function Annotations	3.0	2006
483*	The Theory of Type Hints	–	2014
484*	Type Hints	3.5	2014
482	Literature Overview for Type Hints	–	2015
526*	Syntax for Variable Annotations	3.6	2016
544*	Protocols: Structural subtyping (static duck typing)	3.8	2017
557	Data Classes	3.7	2017
560	Core support for typing module and generic types	3.7	2017
561	Distributing and Packaging Type Information	3.7	2017
563	Postponed Evaluation of Annotations	3.7	2017
586*	Literal Types	3.8	2018
585	Type Hinting Generics In Standard Collections	3.9	2019
589*	TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys	3.8	2019
591*	Adding a final qualifier to typing	3.8	2019
593	Flexible function and variable annotations	?	2019
604	Allow writing union types as X | Y	3.10	2019
612	Parameter Specification Variables	3.10	2019
613	Explicit Type Aliases	3.10	2020

Окончание табл. 15.1

PEP	Название	Python	Год
645	Allow writing optional types as x?	?	2020
646	Variadic Generics	?	2020
647	User-Defined Type Guards	3.10	2021
649	Deferred Evaluation Of Annotations Using Descriptors	?	2021
655	Marking individual TypedDict items as required or potentially missing	?	2021

Официальная документация по Python за всем этим не поспевает, поэтому документация по Муру может служить необходимым справочным руководством. Книга Patrick Viafore «Robust Python» (*https://www.oreilly.com/library/view/robust-python/9781098100650/*) (O'Reilly) стала первым известным мне подробным изложением статической системы типов в Python; она вышла в августе 2021 года. Вторую книгу вы сейчас читаете.

Сложной теме вариантности посвящен отдельный раздел в PEP 484 (*https://peps.python.org/pep-0484/#covariance-and-contravariance*), она также рассмотрена на странице «Обобщенные типы» (*https://mypy.readthedocs.io/en/stable/ generics.html#variance-of-generic-types*) документации по Муру и на неоценимой странице «Типичные проблемы» (*https://mypy.readthedocs.io/en/stable/ common_issues.html#variance*) там же.

Документ PEP 362 «Function Signature Object» (*https://peps.python.org/pep-0362/*) стоит почитать, если вы намереваетесь пользоваться модулем `inspect`, который дополняет функцию `typing.get_type_hints`.

Интересующимся историей Python, возможно, будет полезно знать о сообщении «Adding Optional Static Typing to Python» (*https://www.artima.com/weblogs/viewpost.jsp?thread=85551*), которое Гвидо ван Россум опубликовал 23 декабря 2004 года.

«Python 3 Types in the Wild: A Tale of Two Type Systems» (*https://dl.acm.org/action/cookieAbsent*) – научная статья, написанная Ингкаратом Рак-амноукитом и другими авторами из Ренселаерского политехнического института и Исследовательского центра IBM TJ Watson. В этой работе дан обзор использования аннотаций типов в проектах с открытым исходным кодом и показано, что в большинстве проектов они не используются, а также что в большинстве проектов, где аннотации все-таки используются, не применяется никакое средство проверки типов. Мне показалось очень интересным обсуждение различной семантики Муру и программы *pytype* от Google; по словам авторов, это «по существу две разные системы типов».

Две основополагающие работы по постепенной типизации – Gilad Bracha «Pluggable Type Systems» (*http://bracha.org/pluggableTypesPosition.pdf*) и Eric Meijer and Peter Drayton «Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages» (*https://www.researchgate.net/publication/213886116_Static_Typing_Where_Possible_Dynamic_Typing_When_Needed_The_End_of_the_Cold_War_Between_Programming_Languages*)[1].

[1] Читающие сноски, наверное, вспомнят, что я уже упоминал Эрика Мейера, когда описывал аналогию со школьной столовой при обсуждении вариантности.

Я многому научился, читая относящиеся к теме части книг о других языках программирования, в которых реализованы те же идеи:

- Bruce Eckel, Svetlana Isakova «Atomic Kotlin» (*https://www.atomickotlin.com/atomickotlin/*) (Mindview);
- Joshua Bloch «Effective Java», 3-е издание (Addison-Wesley);
- Vlad Riscutia «Programming with Types: TypeScript Examples» (*https://www.manning.com/books/programming-with-types*) (Manning);
- Boris Cherny «Programming TypeScript» (*https://www.oreilly.com/library/view/programming-typescript/9781492037644/*) (O'Reilly);
- Gilad Bracha «The Dart Programming Language» (Addison-Wesley)[1].

Желающим познакомиться с критическим взглядом на системы типов рекомендую статьи Виктора Юдайкена «Bad ideas in type theory» (*https://www.yodaiken.com/2017/09/15/bad-ideas-in-type-theory/*) и «Types considered harmful II» (*https://www.yodaiken.com/2017/11/30/types-considered-harmful-ii/*).

Наконец, я с удивлением обнаружил статью «Generics Considered Harmful» (*https://web.archive.org/web/20071010002142/http://weblogs.java.net/blog/arnold/archive/2005/06/generics_consid_1.html*) Кена Арнольда, одного из соразработчиков ядра Java с самого начала, а также соавтора первых четырех изданий официального руководства «The Java Programming Language» (Addison-Wesley) – вместе с Джеймсом Гослингом, главным проектировщиком Java.

Как это ни печально, критические замечания Арнольда относятся и к статической системе типов в Python. Читая многочисленные правила и особые случаи в документах PEP, посвященных типизации, я постоянно вспоминал следующий пассаж из статьи Гослинга:

> Что приводит к проблеме, о которой я всегда говорю применительно к C++ и называю ее «исключение порядка N из исключения из правила». Звучит она так: «Мы можем сделать x, но только не в случае y, если y не делает z, в каковом случае это можно сделать, если ...».

По счастью, у Python есть важное преимущество по сравнению Java и C++: его система типов факультативна. Мы можем заткнуть рот программам проверки типов и опустить аннотации типов, если они становятся слишком громоздкими.

Поговорим

Кроличьи норы типизации

Используя программы проверки типов, мы иногда вынуждены открывать для себя и импортировать классы, о которых и знать-то не хотели и которые нашему коду вовсе не нужны – разве что для написания аннотаций типов. Такие классы не документированы, возможно, потому что авторы пакетов считали их деталью реализации. Вот два примера из стандартной библиотеки.

[1] Эта книга была написана о Dart 1. В Dart 2 внесены существенные изменения, в т. ч. в систему типов. Тем не менее Брача – авторитетный исследователь в области проектирования языков программирования, и мне эта книга понравилась за его взгляд на проектирование Dart.

Чтобы воспользоваться функцией `cast()` в примере `server.sockets` из раздела «Приведение типов», я вынужден был прошерстить обширную документацию по *asyncio*, а затем просмотреть исходный код нескольких модулей и пакета, чтобы обнаружить недокументированный класс `TransportSocket`, входящий в состав недокументированного же модуля `asyncio.trsock`. Использовать `socket.socket` вместо `TransportSocket` было бы неправильно, потому что второй не является подтипом первого, как следует из строки документации (*https://github.com/python/cpython/blob/3e7ee02327db13e4337374597cdc4458ecb9e3ad/Lib/asyncio/trsock.py#L5*) в исходном коде.

Я провалился в кроличью нору, когда добавлял аннотации типов в примере 19.13, простой демонстрации пакета `multiprocessing`. Там я использовал объекты типа `SimpleQueue`, которые возвращают вызов `multiprocessing.SimpleQueue()`. Однако я не мог использовать это имя в аннотации типа, потому что оказалось, что `multiprocessing.SimpleQueue` – не класс вовсе! Это связанный метод недокументированного класса `multiprocessing.BaseContext`, который строит и возвращает экземпляр класса `SimpleQueue` в недокументированном модуле `multiprocessing.queues`.

В обоих случаях я вынужден был потратить пару часов, чтобы понять, какой недокументированный класс импортировать, – и все только ради написания одной аннотации типа. Такие изыскания – часть работы автора книги. Но когда я пишу код приложения, я предпочел бы обойтись без такой «охоты за предметами» ради одной непокорной строки и просто написать `# type: ignore`. Иногда это единственное экономически оправданное решение.

Нотация вариантности в других языках

Вариантность – трудная тема, и аннотации типов в Python не так хороши, как могли бы быть. Это подтверждает прямая цитата из PEP 484:

> Ковариантность или контравариантность – свойство не переменной-типа, а обобщенного класса, определенного с использованием этой переменной[1].

Коли так, то почему же ковариантность и контравариантность объявляются с помощью `TypeVar`, а не в обобщенном классе?

Авторы PEP 484 работали в условиях добровольно наложенного на себя ограничения: аннотации типов должны поддерживаться без внесения изменений в интерпретатор. Для этого потребовалось ввести тип `TypeVar` для определения переменных-типов, а также злоупотребить скобками `[]`, чтобы предоставить синтаксис `Klass[T]` для обобщенных типов – вместо нотации `Klass<T>`, применяемой в других популярных языках, включая C#, Java, Kotlin и TypeScript. Ни в одном из этих языков не требуется объявлять переменные-типы до использования.

Кроме того, синтаксис Kotlin и C# ясно дает понять, является ли параметр-тип ковариантным, контравариантным или инвариантным, и именно там, где это нужно: в объявлении класса или интерфейса.

В Kotlin мы могли бы объявить класс `BeverageDispenser` следующим образом:

[1] См. последний абзац раздела «Ковариантность и контравариантность» (*https://peps.python.org/pep-0484/#covariance-and-contravariance*) в документе PEP 484.

```
class BeverageDispenser<out T> {
    // и т. д.
}
```

Модификатор `out` в формальном параметре-типе означает, что `T` – «выходной» тип, а значит, `BeverageDispenser` ковариантен. Вы, наверное, догадались, как можно было бы объявить `TrashCan`:

```
class TrashCan<in T> {
// и т. д.
}
```

Поскольку `T` – «входной» формальный параметр, класс `TrashCan` контравариантен.

Если нет ни `in`, ни `out`, то класс инвариантен относительно параметра.

«Эвристические правила вариантности» вспоминаются, когда в формальных параметрах-типах присутствуют модификаторы `out` или `in`.

Это наводит на мысль, что в Python было бы уместно принять следующие соглашения об именовании ковариантных и контравариантных параметров-типов:

```
T_out = TypeVar('T_out', covariant=True)
T_in = TypeVar('T_in', contravariant=True)
```

Тогда мы могли бы следующим образом определять классы:

```
class BeverageDispenser(Generic[T_out]):
    ...
class TrashCan(Generic[T_in]):
    ...
```

Может быть, еще не поздно изменить соглашение об именовании, предложенное в PEP 484?

Глава 16

Перегрузка операторов

> Есть вещи, которые меня смущают, например перегрузка операторов. Я принял волевое решение исключить перегрузку операторов из языка, потому что видел много примеров злоупотребления этой возможностью в C++.
>
> – Джеймс Гослинг, создатель Java[1]

В Python сложные проценты можно вычислить по следующей формуле:

```
interest = principal * ((1 + rate) ** periods - 1)
```

Операторы, располагающиеся между операндами, как в выражении `1 + rate`, называются *инфиксными*. В Python инфиксные операторы могут применяться к произвольным типам. Таким образом, при работе с вещественными денежными величинами можно гарантировать, что `principal`, `rate` и `periods` будут точными числами, сделав их экземплярами класса `decimal.Decimal`, и формула будет работать, как написано, и давать точный результат.

Но в Java, если перейти от `float` к `BigDecimal` ради получения точных результатов, инфиксными операторами уже нельзя будет воспользоваться, потому что они применимы только к примитивным типам. Та же формула для работы с числами типа `BigDecimal` в Java выглядит так:

```
BigDecimal interest = principal.multiply(BigDecimal.ONE.add(rate)
                        .pow(periods).subtract(BigDecimal.ONE));
```

Ясно, что благодаря инфиксным операторам читать формулы гораздо проще. Для поддержки нотации инфиксных операторов применительно к пользовательским типам или типам расширения, таким как массивы в NumPy, необходима перегрузка операторов. Наличие перегрузки операторов в удобном для работы языке высокого уровня, возможно, и было одной из основных причин поразительного успеха Python как языка обработки данных, в т. ч. для создания финансовых и научных приложений.

В разделе «Эмуляция числовых типов» главы 1 мы видели тривиальные реализации операторов в наброске класса `Vector`. Методы `__add__` и `__mul__` в примере 1.2 были написаны для того, чтобы показать, как специальные методы поддерживают перегрузку операторов, но в их реализации есть тонкие проблемы, на которые мы тогда не стали обращать внимания. Кроме того, в примере 11.2

[1] Источник: «Семейство языков, производных от C: интервью с Деннисом Ритчи, Бьярном Страуструпом и Джеймсом Гослингом» (*http://www.gotw.ca/publications/c_family_interview.htm*).

мы отметили, что в реализации метода `Vector2d.__eq__` предполагается истинным равенство `Vector(3, 4) == [3, 4]` – иногда это имеет смысл, а иногда нет. Эти вопросы станут предметом настоящей главы.

Мы рассмотрим следующие темы:

- как в Python поддерживаются инфиксные операторы с операндами разных типов;
- использование утиной или гусиной типизации при работе с операндами разных типов;
- специальное поведение операторов сравнения (например, `==`, `>`, `<=`);
- подразумеваемая по умолчанию обработка операторов составного присваивания, например `+=`, и их корректная перегрузка.

Что нового в этой главе

Гусиная типизация – ключевая часть Python, но ABC из пакета `numbers` не поддерживаются статической типизацией, поэтому я изменил пример 16.11, так что теперь в нем используется утиная типизация вместо явного сравнения с типом `numbers.Real` с помощью `isinstance`[1].

Я рассматривал оператор матричного умножения `@` в первом издании как запланированное изменение, когда версия 3.5 еще находилась на уровне альфа. Теперь этот оператор описывается не во врезке, а в основном тексте главы в разделе «Использование @ в качестве инфиксного оператора». Я воспользовался гусиной типизацией, чтобы сделать реализацию метода `__matmul__` более безопасной, чем в первом издании, не жертвуя гибкостью.

В разделе «Дополнительная литература» появилась пара новых ссылок, в т. ч. на статью в блоге Гвидо ван Россума. Также я добавил упоминание о двух библиотеках, которые демонстрируют эффективное использование перегрузки операторов за пределами математики: `pathlib` и `Scapy`.

Основы перегрузки операторов

Перегрузка операторов позволяет определенным пользователем объектам взаимодействовать с инфиксными операторами, например `+` и `|`, и с унарными операторами, например `-` и `~`. Вообще, вызов функции (`()`), доступ к атрибутам (`.`), доступ к элементам и срезам (`[]`) реализованы в Python тоже с помощью операторов, но в этой главе рассматриваются только унарные и инфиксные операторы.

У перегрузки операторов сложилась дурная репутация в некоторых кругах. Это языковое средство, которое легко использовать неправильно (что не раз происходило), а результат – недоумение программиста, ошибки и неожиданные провалы производительности. Зато при правильном употреблении мы получаем приятный API и удобочитаемый код. Python стремится найти баланс между гибкостью, удобством и безопасностью, для чего вводятся некоторые ограничения:

---

[1] Остальные ABC в стандартной библиотеке Python по-прежнему сохраняют ценность для гусиной и статической типизации. Проблема с ABC из пакета `numbers` объясняется в разделе « ABC из пакета numbers и числовые протоколы» главы 13.

- запрещается перегружать операторы для встроенных типов;
- запрещается создавать новые операторы, можно только перегружать существующие;
- несколько операторов перегружать нельзя вовсе: `is`, `and`, `or`, `not` (на поразрядные операторы `&`, `|`, `~` это не распространяется).

В классе `Vector` из главы 12 нам уже встречался инфиксный оператор `==`, поддерживаемый методом `__eq__`. В этой главе мы улучшим реализацию `__eq__`, чтобы правильнее обрабатывать операнды, типы которых отличаются от `Vector`. Однако операторы сравнения (`==`, `!=`, `>`, `<`, `>=`, `<=`) – это особые случаи перегрузки операторов, поэтому начнем с перегрузки четырех арифметических операторов в классе `Vector`: сначала унарных `-` и `+`, а затем инфиксных `+` и `*`.

Начнем с простейшей темы: унарных операторов.

Унарные операторы

В разделе 6.5 «Унарные арифметические и поразрядные операции» (*https://docs.python.org/3/reference/expressions.html#unary-arithmetic-and-bitwise-operations*) справочного руководства по языку Python перечислены три унарных оператора, которые ниже показаны вместе с относящимися к ним специальными методами.

`-` *(реализован с помощью* `__neg__`*)*

Унарный арифметический минус. Если `x` равно `-2`, то `-x == 2`.

`+` *(реализован с помощью* `__pos__`*)*

Унарный арифметический плюс. Обычно `x == +x`, но есть несколько особых случаев, когда это неверно. Если вам интересно, см. врезку «Когда x не равно +x» ниже.

`~` *(реализован с помощью* `__invert__`*)*

Поразрядная инверсия целого числа, определяется как `~x == -(x+1)`. Если `x` равно `2`, то `~x == -3`[1].

В главе «Модель данных» (*https://docs.python.org/3/reference/datamodel.html#object.__neg__*) справочного руководства по языку Python встроенная функция `abs()` также названа унарным оператором. Ранее мы видели, что с ней связан специальный метод `__abs__`.

Поддержать унарные операторы легко. Достаточно реализовать соответствующий специальный метод, который принимает единственный аргумент `self`. Логика этого метода может быть произвольной, но должно удовлетворяться фундаментальное правило: оператор всегда возвращает новый объект. Иначе говоря, не модифицируйте `self`, а создавайте и возвращайте новый экземпляр подходящего типа.

В случае операторов `-` и `+` результат, вероятно, должен быть экземпляром того же класса, что и `self`. Для унарного `+`, если объект, от имени которого вызывается оператор, неизменяемый, то следует возвращать `self`, иначе копию `self`. Для `abs()` результатом должен быть скаляр.

[1] Объяснение поразрядной операции not см. по адресу *https://en.wikipedia.org/wiki/Bitwise_operation#NOT*.

Что касается оператора `~`, то трудно сказать, какой результат считать разумным, если речь не идет о битах целого числа. В пакете для анализа данных *pandas* оператор тильды инвертирует булевы условия фильтрации; примеры см. в разделе «Булева индексация» (*https://pandas.pydata.org/pandas-docs/stable/user_guide/ indexing.html#boolean-indexing*) документации по pandas.

Как и было обещано, мы реализуем еще несколько операторов в классе `Vector` в дополнение к тому, что было сделано в главе 12. В примере 16.1 показан метод `__abs__` – тот же, что в примере 12.16, – и новые методы `__neg__` и `__pos__` для поддержки унарных операторов.

Пример 16.1. vector_v6.py: унарные операторы `-` и `+` в дополнение к примеру 12.16

```
    def __abs__(self):
        return math.hypot(*self)

    def __neg__(self):
        return Vector(-x for x in self) ❶

    def __pos__(self):
        return -Vector(self) ❷
```

❶ Для вычисления `-v` строим новый объект `Vector`, в котором все компоненты `self` имеют противоположный знак.

❷ Для вычисления `+v` строим новый объект `Vector` с точно такими же компонентами, как у `self`.

Напомним, что экземпляры `Vector` – итерируемые объекты, а `Vector.__init__` принимает в качестве аргумента итерируемый объект, поэтому реализации `__neg__` и `__pos__` оказались очень короткими и элегантными.

Мы не станем реализовывать метод `__invert__`, поэтому при попытке выполнить операцию `~v` для экземпляра `Vector` Python возбудит исключение `TypeError` с не оставляющим сомнений сообщением: «bad operand type for unary ~: 'Vector'».

Прочитав об одном курьезе во врезке ниже, вы сможете как-нибудь при случае выиграть пари, касающееся унарного `+`.

Когда x не равно +x

Все ожидают, что `x == +x`, и почти всегда в Python так оно и есть, но я нашел в стандартной библиотеке два случая, когда `x != +x`.

Первый касается класса `decimal.Decimal`. Может получиться, что `x != +x`, если `x` – экземпляр `Decimal`, созданный в арифметическом контексте, а `+x` затем вычислялось в контексте с другими свойствами. Например, `x` вычисляется в контексте с некоторой точностью, затем точность изменяется, после чего вычисляется `+x`. См. демонстрацию в примере 16.2.

Пример 16.2. Изменение точности в арифметическом контексте может привести к тому, что `x` будет отличаться от `+x`

```
>>> import decimal
>>> ctx = decimal.getcontext() ❶
>>> ctx.prec = 40 ❷
```

```
>>> one_third = decimal.Decimal('1') / decimal.Decimal('3') ❸
>>> one_third ❹
Decimal('0.333333333333333333333333333333333333333333333')
>>> one_third == +one_third ❺
True
>>> ctx.prec = 28 ❻
>>> one_third == +one_third ❼
False
>>> +one_third ❽
Decimal('0.3333333333333333333333333333')
```

❶ Получить ссылку на текущий глобальный арифметический контекст.

❷ Установить точность арифметического контекста 40.

❸ Вычислить 1/3 с текущей точностью.

❹ Напечатать результат – имеем 40 цифр после точки.

❺ Значение выражения `one_third == +one_third` равно `True`.

❻ Понизить точность до 28 – значение по умолчанию для класса `Decimal` в Python 3.4.

❼ Теперь значение выражения `one_third == +one_third` равно `False`.

❽ Напечатать `+one_third` – после точки только 28 цифр.

Получается, что при каждом вычислении выражения `+one_third` создается новый экземпляр `Decimal` со значением `one_third`, но с точностью, заданной в текущем арифметическом контексте.

Второй случай, когда `x != +x`, описан в документации по классу `collections.Counter` (*https://docs.python.org/3/library/collections.html#collections.Counter*). В классе `Counter` реализовано несколько арифметических операторов, в том числе инфиксный `+`, который складывает счетчики соответственных элементов из двух объектов `Counter`. Однако из практических соображений сложение в классе `Counter` не включает в результат элементы с отрицательным или нулевым счетчиком. А унарный `+` прибавляет пустой объект `Counter` и, следовательно, сохраняет только те элементы, в которых счетчик больше нуля.

Пример 16.3. Унарный `+` порождает новый объект `Counter`, в который не входят элементы с нулевыми и отрицательными счетчиками

```
>>> ct = Counter('abracadabra')
>>> ct
Counter({'a': 5, 'r': 2, 'b': 2, 'd': 1, 'c': 1})
>>> ct['r'] = -3
>>> ct['d'] = 0
>>> ct
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 0, 'r': -3})
>>> +ct
Counter({'a': 5, 'b': 2, 'c': 1})
```

Как видите, `+ct` возвращает объект `Counter`, в котором все счетчики больше нуля.

А теперь вернемся к обычному программированию.

Перегрузка оператора сложения векторов +

Класс `Vector` – это последовательность, а в разделе 3.3.6 «Эмуляция контейнерных типов» главы «Модель данных» (*https://docs.python.org/3/reference/datamodel.html#emulating-container-types*) говорится, что последовательности должны поддерживать оператор `+` с семантикой конкатенации и оператор `*` с семантикой повторения. Однако в данном случае мы реализуем `+` и `*` как математические операторы, что несколько труднее, но для типа `Vector` более осмысленно.

Если пользователь хочет конкатенировать или повторить экземпляры `Vector`, то может преобразовать их в кортежи или списки, применить оператор и преобразовать обратно – поскольку сам объект `Vector` итерируемый и может быть сконструирован из итерируемого объекта:

```
>>> v_concatenated = Vector(list(v1) + list(v2))
>>> v_repeated = Vector(tuple(v1) * 5)
```

Сложение двух евклидовых векторов дает новый вектор, компоненты которого являются суммами соответственных компонент слагаемых, например:

```
>>> v1 = Vector([3, 4, 5])
>>> v2 = Vector([6, 7, 8])
>>> v1 + v2
Vector([9.0, 11.0, 13.0])
>>> v1 + v2 == Vector([3 + 6, 4 + 7, 5 + 8])
True
```

Что будет, если сложить два экземпляра `Vector` разной длины? Мы могли бы возбудить исключение, но в реальных приложениях (например, в информационном поиске) лучше дополнить более короткий вектор нулями. Вот какой результат мы хотим получить:

```
>>> v1 = Vector([3, 4, 5, 6])
>>> v3 = Vector([1, 2])
>>> v1 + v3
Vector([4.0, 6.0, 5.0, 6.0])
```

При таких требованиях мы можем реализовать `__add__`, как показано в примере 16.4.

Пример 16.4. Метод `Vector.add`, попытка № 1

```
    # в классе Vector

    def __add__(self, other):
        pairs = itertools.zip_longest(self, other, fillvalue=0.0) ❶
        return Vector(a + b for a, b in pairs) ❷
```

❶ `pairs` – генератор, который порождает кортежи `(a, b)`, где `a` берется из `self`, а `b` – из `other`. Если длины `self` и `other` различаются, то более короткий вектор дополняется значениями `fillvalue`.

❷ Новый объект `Vector` инициализируется генераторным выражением, которое порождает по одной сумме для каждого элемента `pairs`.

Обратите внимание, что `__add__` возвращает новый экземпляр `Vector`, не изменяя ни `self`, ни `other`.

Специальные методы, реализующие унарные или инфиксные операторы, не должны изменять свои операнды. Предполагается, что выражения, содержащие такие операторы, вычисляют результаты, создавая новые объекты. И лишь операторы составного присваивания могут изменять свой первый операнд (`self`), о чем речь пойдет ниже.

В примере 16.4 разрешено прибавлять `Vector` к `Vector2d`, а также к кортежу или любому другому итерируемому объекту, порождающему числу. Это доказывает пример 16.5.

Пример 16.5. `Vector.__add__` из примера 16.4 поддерживает сложение с объектами, отличными от `Vector`

```
>>> v1 = Vector([3, 4, 5])
>>> v1 + (10, 20, 30)
Vector([13.0, 24.0, 35.0])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v1 + v2d
Vector([4.0, 6.0, 5.0])
```

Оба сложения в примере 16.5 работают, потому что в методе `__add__` используется функция `zip_longest(…)`, готовая принимать любые итерируемые объекты, а генераторное выражение, которым инициализируется новый `Vector`, просто выполняет операцию `a + b` для каждой пары, возвращаемой `zip_longest(…)`, поэтому подойдет любой итерируемый объект, порождающий числа.

Однако если поменять операнды местами (пример 16.6), то сложение операндов разных типов даст ошибку.

Пример 16.6. `Vector.__add__` из примера 16.4 дает ошибку, если тип левого операнда – не `Vector`

```
>>> v1 = Vector([3, 4, 5])
>>> (10, 20, 30) + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "Vector") to tuple
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v2d + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vector2d' and 'Vector'
```

Для поддержки операций с объектами разных типов в Python имеется особый механизм диспетчеризации для специальных методов, ассоциированных с инфиксными операторами. Видя выражение `a + b`, интерпретатор выполняет следующие шаги (см. рис. 16.1).

1. Если у `a` есть метод `__add__`, вызвать `a.__add__(b)` и вернуть результат, если только он не равен `NotImplemented`.

2. Если у `a` нет метода `__add__` или его вызов вернул `NotImplemented`, проверить, есть ли у `b` метод `__radd__`, и, если да, вызвать `b.__radd__(a)` и вернуть результат, если только он не равен `NotImplemented`.
3. Если у `b` нет метода `__radd__` или его вызов вернул `NotImplemented`, возбудить исключение `TypeError` с сообщением *unsupported operand types* (неподдерживаемые типы операндов).

Метод `__radd__` называется «инверсным» (reversed), или «отраженным» (reflected), вариантом `__add__`. Я предпочитаю термин «инверсные» специальные методы[1].

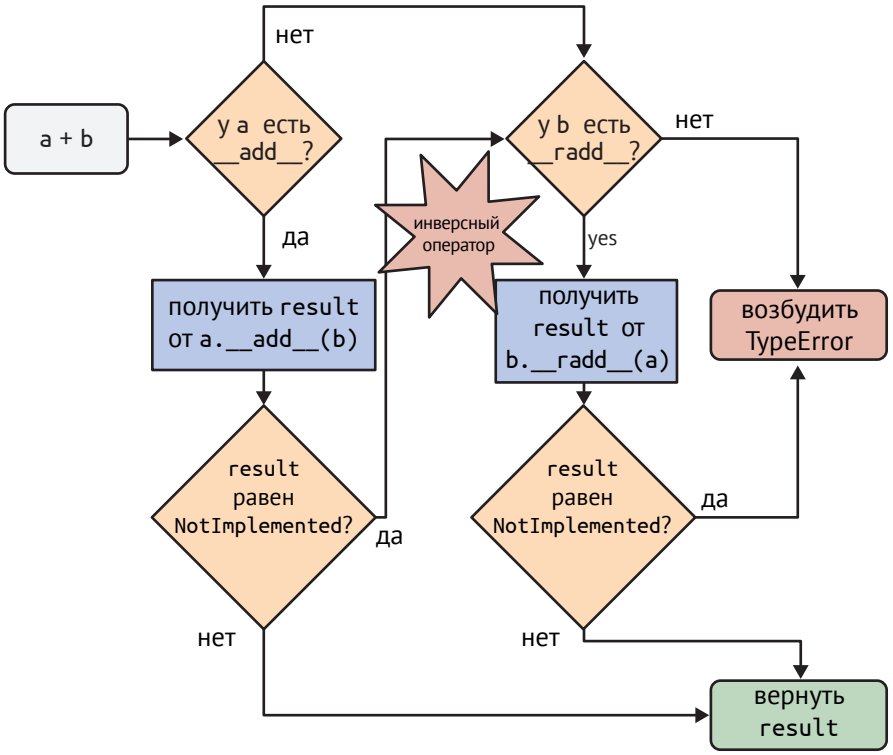


Рис. 16.1. Блок-схема вычисления `a + b` с помощью перегруженных операторов `__add__` и `__radd__`

[1] В документации по Python встречаются оба термина. В главе «Модель данных» (*https://docs.python.org/3/reference/datamodel.html*) используется «reflected», а в разделе 9.1.2.2 «Реализация арифметических операций» (*https://docs.python.org/3/library/numbers.html#implementing-the-arithmetic-operations*) при описании модуля `numbers` упоминаются «forward» (прямые) и «reverse» (инверсные) методы, и мне эта терминология нравится больше, потому что слова «прямой» и «инверсный» (не «обратный», чтобы не путать с обратной функцией. – *Прим. перев.*) сразу наводят на мысль о направлении, тогда как у слова «reflected» нет очевидного антонима.

Итак, чтобы сложение операндов разных типов в примере 16.6 заработало, мы должны реализовать метод `Vector.__radd__`, который Python вызовет, если у левого операнда нет метода `__add__` или есть, но возвращает значение `NotImplemented`, сигнализируя о том, что не знает, как обработать правый операнд.

Не путайте `NotImplemented` с `NotImplementedError`. `NotImplemented` – это значение-синглтон, которое должен возвращать специальный метод инфиксного оператора, чтобы сообщить интерпретатору о том, что не умеет обрабатывать данный операнд. А `NotImplementedError` – исключение, которое возбуждают методы-заглушки в абстрактных классах, предупреждая, что их необходимо переопределить в подклассах.

Простейшая работающая реализация метода `__radd__` показана в примере 16.7.

Пример 16.7. Методы `Vector.__add__` и `__radd__`

```
    # в классе Vector

    def __add__(self, other): ❶
        pairs = itertools.zip_longest(self, other, fillvalue=0.0)
        return Vector(a + b for a, b in pairs)

    def __radd__(self, other): ❷
        return self + other
```

❶ Метод `__add__` такой же, как в примере 16.4; приведен только потому, что им пользуется метод `__radd__`.

❷ `__radd__` просто делегирует свою работу методу `__add__`.

Часто инверсный оператор можно таким и оставить: просто делегировать работу нужному оператору, в данном случае `__add__`. Это относится к любому коммутативному оператору; `+` является коммутативным для чисел и векторов, но перестает быть таковым, когда используется для конкатенации последовательностей в Python.

Если `__radd__` просто вызывает `__add__`, то того же результата можно достичь и другим способом:

```
    def __add__(self, other):
        pairs = itertools.zip_longest(self, other, fillvalue=0.0)
        return Vector(a + b for a, b in pairs)

    __radd__ = __add__
```

Методы в примере 16.7 работают как с объектами `Vector`, так и с любыми другими итерируемыми объектами, содержащими числовые элементы: `Vector2d`, кортеж целых чисел или массив чисел с плавающей точкой. Но если методу `__add__` подсунуть неитерируемый объект, то он выдаст не слишком полезное сообщение об ошибке, как в примере 16.8.

Пример 16.8. Методу `Vector.__add__` необходим итерируемый операнд

```
>>> v1 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "vector_v6.py", line 328, in __add__
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
TypeError: zip_longest argument #2 must support iteration
```

Хуже того, мы получаем сбивающее с толку сообщение, если операнд – итерируемый объект, но его элементы нельзя сложить с элементами `Vector`, имеющими тип `float`. См. пример 16.9.

Пример 16.9. Методу `Vector.__add__` необходим итерируемый операнд с числовыми элементами

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs)
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components)
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Я пытался сложить `Vector` и `str`, но в сообщении упоминаются `float` и `str`.

Однако непонятные сообщения в примерах 16.8 и 16.9 – еще не самое страшное: если специальный метод оператора не может вернуть правильный результат из-за несовместимости типов, он должен возвращать значение `NotImplemented`, а не возбуждать исключение `TypeError`. Возвращая `NotImplemented`, вы оставляете разработчику типа другого операнда возможность выполнить операцию, когда Python попробует вызвать инверсный метод.

Оставаясь верны духу утиной типизации, мы воздержимся от проверки типа операнда `other` или его элементов. Вместо этого мы перехватим исключение и вернем `NotImplemented`. Если интерпретатор еще не пробовал операнды в обратном порядке, то сделает это. Если же значение `NotImplemented` вернул инверсный метод, то Python возбудит исключение `TypeError` со стандартным сообщением вида «unsupported operand type(s) for +: *Vector* and *str*».

Окончательная реализация специальных методов для сложения объектов класса `Vector` приведена в примере 16.10.

Пример 16.10. vector_v6.py: специальные методы оператора `+`, добавленные в файл vector_v5.py (пример 12.16)

```
    def __add__(self, other):
        try:
            pairs = itertools.zip_longest(self, other, fillvalue=0.0)
            return Vector(a + b for a, b in pairs)
        except TypeError:
            return NotImplemented

    def __radd__(self, other):
        return self + other
```

Заметим, что теперь `__add__` перехватывает `TypeError` и возвращает `NotImplemented`.

Если метод инфиксного оператора возбуждает исключение, то работа алгоритма диспетчеризации прерывается. В частном случае исключения `TypeError` зачастую лучше перехватить его и вернуть значение `NotImplemented`. Это позволит интерпретатору вызвать метод инверсного оператора, который, возможно, сумеет завершить вычисление, поменяв местами операнды разных типов.

Итак, мы безопасно перегрузили оператор `+`, написав методы `__add__` и `__radd__`. Теперь займемся инфиксным оператором `*`.

Перегрузка оператора умножения на скаляр *

Что означает запись `Vector([1, 2, 3]) * x`? Если `x` – число, то это умножение на скаляр, результатом которого является новый объект `Vector`, каждая компонента которого является произведением `x` и соответственной компоненты исходного вектора:

```
>>> v1 = Vector([1, 2, 3])
>>> v1 * 10
Vector([10.0, 20.0, 30.0])
>>> 11 * v1
Vector([11.0, 22.0, 33.0])
```

Над векторами определена и другая операция умножения: скалярное произведение; если представить один вектор как матрицу 1×N, а другой – как матрицу N×1, то результат перемножения этих матриц и называется скалярным произведением. Мы реализуем этот оператор в классе `Vector` в разделе «Использование @ как инфиксного оператора».

Но вернемся к операции умножения на скаляр. Как и раньше, начнем с простейших вариантов `__mul__` и `__rmul__`:

```
    # в классе Vector

    def __mul__(self, scalar):
        return Vector(n * scalar for n in self)

    def __rmul__(self, scalar):
        return self * scalar
```

Оба метода работают, если типы операндов совместимы. Аргумент `scalar` должен быть числом, которое при умножении на `float` дает `float` (поскольку во внутреннем представлении класса `Vector` используется массив чисел типа `float`). Поэтому число типа `complex` не подойдет, однако годятся типы `int`, `bool` (поскольку `bool` – подкласс `int`) и даже `fractions.Fraction`. В примере 16.11 метод `__mul__` не проверяет тип `scalar` явно, а пытается преобразовать его в тип `float` и возвращает `NotImplemented`, если эта попытка завершается неудачно. Это яркий пример утиной типизации.

Пример 16.11. vector_v7.py: добавлены методы оператора *

```
class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    # много методов опущено, полный код см. в файле vector_v7.py
    # по адресу https://github.com/fluentpython/example-code-2e

    def __mul__(self, scalar):
        try:
            factor = float(scalar)
        except TypeError: ❶
            return NotImplemented ❷
        return Vector(n * factor for n in self)

    def __rmul__(self, scalar):
        return self * scalar ❸
```

❶ Если `scalar` нельзя преобразовать в `float`...

❷ ... то мы не знаем, как его обработать, поэтому возвращаем `NotImplemented`, чтобы Python мог попробовать оператор `__rmul__` операнда `scalar`.

❸ В этом примере `__rmul__` просто вычисляет произведение `self * scalar`, делегируя всю работу методу `__mul__`.

Код из примера 16.11 позволяет умножать векторы на скалярные значения обычных и не очень обычных числовых типов:

```
>>> v1 = Vector([1.0, 2.0, 3.0])
>>> 14 * v1
Vector([14.0, 28.0, 42.0])
>>> v1 * True
Vector([1.0, 2.0, 3.0])
>>> from fractions import Fraction
>>> v1 * Fraction(1, 3)
Vector([0.3333333333333333, 0.6666666666666666, 1.0])
```

Научившись умножать `Vector` на скаляры, посмотрим, как реализовать умножение `Vector` на `Vector`.

В первом издании я использовал в примере 16.11 гусиную типизацию: проверял аргумент `scalar` метода `__mul__` с помощью `isinstance(scalar, numbers.Real)`. Теперь я избегаю ABC из пакета `numbers`, потому что они не поддерживаются документом PEP 484, а использовать во время выполнения типы, которые нельзя проверить статически, мне не нравится.

Альтернативно я мог бы проверять поддержку протокола `typing.SupportsFloat`, как мы видели в разделе «Статические протоколы, допускающие проверку во время выполнения» главы 13. Я выбрал в этом примере утиную типизацию, потому что считаю, что мастеровитый питонист должен свободно владеть этим приемом кодирования.

С другой стороны, метод `__matmul__` в примере 16.12, добавленном в этом издании, дает хороший пример гусиной типизации.

Использование @ как инфиксного оператора

Знак `@` хорошо известен как префикс декораторов функций, но начиная с 2015 года он используется также в качестве инфиксного оператора. В течение многих лет скалярное произведение записывалось в виде `numpy.dot(a, b)` в NumPy. Синтаксис вызова функции усложняет запись длинных математических формул на Python[1], поэтому часть сообщества, занимающаяся численными методами, пролоббировала документ PEP 465 «A dedicated infix operator for matrix multiplication» (*https://peps.python.org/pep-0465/*), который был реализован в Python 3.5. Сегодня мы можем писать `a @ b`, когда нужно вычислить скалярное произведение двух массивов NumPy.

Оператор `@` поддерживается специальными методами `__matmul__`, `__rmatmul__` и `__imatmul__`, имена которых означают «matrix multiplication» (умножение матриц). В настоящее время эти методы не используются в стандартной библиотеке, но интерпретатор знает о них, начиная с версии Python 3.5, так что команда NumPy, да и все мы можем поддержать оператор `@` в определенных пользователем типах. Синтаксический анализатор также был изменен и теперь умеет обрабатывать новый оператор (конструкция `a @ b` в Python 3.4 считалась синтаксической ошибкой).

Следующие простые тесты показывают, как `@` должен работать с экземплярами `Vector`:

```
>>> va = Vector([1, 2, 3])
>>> vz = Vector([5, 6, 7])
>>> va @ vz == 38.0 # 1*5 + 2*6 + 3*7
True
>>> [10, 20, 30] @ vz
380.0
>>> va @ 3
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for @: 'Vector' and 'int'
```

В примере 16.12 показан код необходимых специальных методов.

Пример 16.12. vector_v7.py: методы оператора `@`

```
class Vector:
    # много методов опущено

    def __matmul__(self, other):
        if (isinstance(other, abc.Sized) and ❶
            isinstance(other, abc.Iterable)):
            if len(self) == len(other): ❷
                return sum(a * b for a, b in zip(self, other)) ❸
            else:
                raise ValueError('@ requires vectors of equal length.')
        else:
            return NotImplemented

    def __rmatmul__(self, other):
        return self @ other
```

[1] См. обсуждение этой проблемы в разделе «Поговорим».

❶ Оба операнда должны реализовывать методы `__len__` и `__iter__` …
❷ … и иметь одинаковую длину, чтобы можно было …
❸ … элегантно применить `sum`, `zip` и генераторное выражение.

Новая возможность zip() в Python 3.10

Начиная с версии Python 3.10 встроенная функция `zip` принимает факультативный чисто именованный аргумент `strict`. Если `strict=True`, то функция возбуждает исключение `ValueError`, когда итерируемые объекты имеют разную длину. По умолчанию `strict` равен `False`. Это новое поведение согласуется с принципом быстрого отказа, принятым в Python. В примере 16.12 я бы заменил внутреннее предложение `if` предложением `try/except ValueError` и добавил при вызове `zip` аргумент `strict=True`.

Пример 16.12 – хорошая иллюстрация практического применения гусиной типизации. Если бы мы сравнивали тип операнда `other` с `Vector`, то лишили бы пользователей возможности использовать списки или массивы в качестве операндов `@`. Коль скоро один операнд имеет тип `Vector`, наша реализация `@` поддерживает другие операнды, являющиеся экземплярами `abc.Sized` и `abc.Iterable`. Оба этих ABC реализуют метод `__subclasshook__`, поэтому любой объект, предоставляющий методы `__len__` и `__iter__`, проходит наш тест – и не нужно ни наследовать этим ABC, ни даже регистрировать их виртуальные подклассы (см. раздел «ABC и структурная типизация» главы 13). В частности, наш класс `Vector` не наследует ни `abc.Sized`, ни `abc.Iterable`, но проходит проверки на совместимость с этими ABC с помощью `isinstance`, потому что имеет необходимые методы.

Арифметические операторы – итоги

При реализации операторов `+` и `*` мы познакомились с наиболее распространенными приемами программирования инфиксных операторов. Описанная техника применима ко всем операторам, перечисленным в табл. 16.1 (операторы, вычисляемые на месте, будут рассмотрены в разделе «Составные операторы присваивания» ниже).

Таблица 13.1. Имена методов инфиксных операторов (операторы, вычисляемые на месте, связаны с составным присваиванием; операторы сравнения описаны в табл. 16.2)

Оператор	Прямой	Инверсный	На месте	Описание
`+`	`__add__`	`__radd__`	`__iadd__`	Сложение или конкатенация
`-`	`__sub__`	`__rsub__`	`__isub__`	Вычитание
`*`	`__mul__`	`__rmul__`	`__imul__`	Умножение или повторение
`/`	`__truediv__`	`__rtruediv__`	`__itruediv__`	Истинное деление
`//`	`__floordiv__`	`__rfloordiv__`	`__ifloordiv__`	Деление с округлением
`%`	`__mod__`	`__rmod__`	`__imod__`	Деление по модулю
`divmod()`	`__divmod__`	`__rdivmod__`	`__idivmod__`	Возвращает кортеж, содержащий частное и остаток

Окончание табл. 13.1

Оператор	Прямой	Инверсный	На месте	Описание
`**, pow()`	`__pow__`	`__rpow__`	`__ipow__`	Возведение в степень[a]
`@`	`__matmul__`	`__rmatmul__`	`__imatmul__`	Матричное умножение
`&`	`__and__`	`__rand__`	`__iand__`	Поразрядное И
`|`	`__or__`	`__ror__`	`__ior__`	Поразрядное ИЛИ
`^`	`__xor__`	`__rxor__`	`__ixor__`	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ
`<<`	`__lshift__`	`__rlshift__`	`__ilshift__`	Поразрядный сдвиг влево
`>>`	`__rshift__`	`__rrshift__`	`__irshift__`	Поразрядный сдвиг вправо

[a] Оператор `pow` принимает необязательный третий аргумент, `modulo`: `pow(a, b, modulo)`, поддерживаемый также специальными методами, если они вызываются напрямую (например, `a.__pow__(b, modulo)`).

Для операторов сравнения действуют несколько иные правила.

Операторы сравнения

Обработка операторов сравнения `==`, `!=`, `>`, `<`, `>=`, `<=` интерпретатором Python похожа на то, что мы видели выше, но имеет два важных отличия.

- Для прямых и инверсных вызовов служит один и тот же набор методов. Правила приведены в табл. 16.2. Например, в случае оператора `==` как прямой, так и инверсный вызовы обращаются к методу `__eq__`, но изменяется порядок аргументов. А прямой вызов `__gt__` сопровождается инверсным вызовом `__lt__` с переставленными аргументами.
- В случае `==` и `!=`, если инверсный метод отсутствует или возвращает `NotImplemented`, Python сравнивает идентификаторы объектов, а не возбуждает исключение `TypeError`.

Таблица 16.2. Операторы сравнения: инверсные методы вызываются, когда первый вызов вернул NotImplemented

Группа	Инфиксный оператор	Прямой вызов метода	Инверсный вызов метода	Запасной вариант
Равенство	a == b	a.__eq__(b)	b.__eq__(a)	Вернуть id(a) == id(b)
	a != b	a.__ne__(b)	b.__ne__(a)	Вернуть not (a == b)
Порядок	a > b	a.__gt__(b)	a.__lt__(b)	Возбудить `TypeError`
	a < b	a.__lt__(b)	a.__gt__(b)	Возбудить `TypeError`
	a >= b	a.__ge__(b)	a.__le__(b)	Возбудить `TypeError`
	a <= b	a.__le__(b)	a.__ge__(b)	Возбудить `TypeError`

Имея в виду эти правила, давайте улучшим поведение метода `Vector.__eq__`, которое в файле *vector_v5.py* (пример 12.16) было закодировано следующим образом:

```
    class Vector:
        # много строк опущено

        def __eq__(self, other):
            return (len(self) == len(other) and
                    all(a == b for a, b in zip(self, other)))
```

В примере 16.13 показаны результаты работы этого метода.

Пример 16.13. Сравнение `Vector` с `Vector`, с `Vector2d` и с `tuple`

```
>>> vb = Vector(range(1, 4))
>>> va == vb ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3 ❸
True
```

❶ Два объекта `Vector` с равными числовыми компонентами должны быть равны.
❷ Объекты `Vector` и `Vector2d` также равны, если равны их компоненты.
❸ `Vector` считается равным кортежу или любому другому итерируемому объекту, с числовыми элементами, соответственно равными его элементам.

Последний результат в примере 13.12 вряд ли следует считать желательным. Действительно ли мы хотим, чтобы `Vector` считался равным кортежу, содержащему такие же числа? Впрочем, твердой уверенности у меня нет – все зависит от контекста. Однако в «Дзен Python» сказано:

> Встретив неоднозначность, отбрось искушение угадать.

Излишняя либеральность при вычислении операндов может преподнести сюрпризы, а программисты их ненавидят.

Если в поисках ключа обратиться к самому Python, то мы увидим, что сравнение `[1,2] == (1, 2)` дает `False`. Поэтому будем осторожны и добавим сравнение типов. Если второй операнд – объект класса `Vector` (или его подкласса), то оставим ту же логику, что в текущей реализации `__eq__`. Иначе вернем `NotImplemented`, и пусть Python разбирается.

Пример 16.14. vector_v8.py: улучшенный метод `__eq__` в классе `Vector`

```
    def __eq__(self, other):
        if isinstance(other, Vector): ❶
            return (len(self) == len(other) and
                    all(a == b for a, b in zip(self, other)))
        else:
            return NotImplemented ❷
```

❶ Если операнд `other` – объект класса `Vector` (или его подкласса), то выполнить сравнение, как и раньше.
❷ Иначе вернуть `NotImplemented`.

Прогнав тесты из примера 16.13 для новой реализации `Vector.__eq__`, мы получим следующие результаты.

Пример 16.15. Те же сравнения, что в примере 16.13, последний результат изменился

```
>>> vb = Vector(range(1, 4))
>>> va == vb ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3 ❸
False
```

❶ Тот же результат, что и раньше. Как и ожидалось.
❷ Тот же результат, что и раньше. Но почему? Объяснение последует ниже.
❸ Другой результат – то, что мы и хотели. Но почему это работает? Читайте дальше…

Из трех результатов в примере 16.15 первый не вызывает удивления, а два других объясняются тем, что метод `__eq__` из примера 16.14 вернул `NotImplemented`. Рассмотрим шаг за шагом, что происходит при сравнении `Vector` и `Vector2d`.

1. Для вычисления `vc == v2d` Python вызывает `Vector.__eq__(vc, v2d)`.
2. Метод `Vector.__eq__(vc, v2d)` видит, что `v2d` не принадлежит классу `Vector`, и возвращает `NotImplemented`.
3. Получив результат `NotImplemented`, Python вызывает `Vector2d.__eq__(v2d, vc)`.
4. `Vector2d.__eq__(v2d, vc)` преобразует оба операнда в кортежи и сравнивает их, результат оказывается равен `True` (код метода `Vector2d.__eq__` приведен в примере 11.11).

При сравнении же `Vector` и `tuple` производятся следующие шаги.

1. Для вычисления `va == t3` Python вызывает `Vector.__eq__(va, t3)`.
2. Метод `Vector.__eq__(va, t3)` видит, что `t3` не принадлежит классу `Vector`, и возвращает `NotImplemented`.
3. Получив результат `NotImplemented`, Python вызывает `tuple.__eq__(t3, va)`.
4. `tuple.__eq__(t3, va)` ничего не знает о классе `Vector`, поэтому возвращает `NotImplemented`.
5. Оператор `==` рассматривается как особый случай: если инверсный вызов вернул `NotImplemented`, то Python в качестве последнего средства сравнивает идентификаторы объектов.

Нам не нужно реализовывать метод `!=`, потому что поведение метода `__ne__`, унаследованное от `object`, нас вполне устраивает: если `__eq__` определен и возвращает что-то, кроме `NotImplemented`, то `__ne__` возвращает противоположное значение.

Иными словами, при тех же объектах, что в примере 16.15, результаты оператора `!=` непротиворечивы:

```
>>> va != vb
False
>>> vc != v2d
False
>>> va != (1, 2, 3)
True
```

Метод `__ne__`, унаследованный от `object`, работает, как показано в следующем фрагменте, хотя в действительности он написан на C[1]:

```
    def __ne__(self, other):
        eq_result = self == other
        if eq_result is NotImplemented:
            return NotImplemented
        else:
            return not eq_result
```

Рассмотрев перегрузку инфиксных операторов, обратимся к операторам составного присваивания.

Операторы составного присваивания

Наш класс `Vector` уже поддерживает операторы составного присваивания `+=` и `*=`. Это объясняется тем, что для неизменяемых объектов оператор составного присваивания создает новый экземпляр и связывает его с переменной в левой части.

В примере 16.16 операторы составного присваивания показаны в действии.

Пример 16.16. Использование += и *= для экземпляров `Vector`

```
>>> v1 = Vector([1, 2, 3])
>>> v1_alias = v1 ❶
>>> id(v1) ❷
4302860128
>>> v1 += Vector([4, 5, 6]) ❸
6
>>> v1 ❹
Vector([5.0, 7.0, 9.0])
>>> id(v1) ❺
4302859904
>>> v1_alias ❻
Vector([1.0, 2.0, 3.0])
>>> v1 *= 11 ❼
>>> v1 ❽
Vector([55.0, 77.0, 99.0])
>>> id(v1)
4302858336
```

❶ Создать псевдоним, чтобы можно было проинспектировать объект `Vector([1, 2, 3])` позже.

[1] Логика методов `object.__eq__` и `object.__ne__` для интерпретатора CPython реализована в функции `object_richcompare` в исходном файле *Objects/typeobject.c* (*https://github.com/python/cpython/blob/0bbf30e2b910bc9c5899134ae9d73a8df968da35/Objects/typeobject.c#L4598*).

❷ Запомнить идентификатор исходного объекта `Vector`, связанного с `v1`.
❸ Выполнить составное сложение.
❹ Результат ожидаемый…
❺ …но создан новый `Vector`.
❻ Проинспектировать `v1_alias`, чтобы убедиться, что исходный `Vector` не изменился.
❼ Выполнить составное умножение.
❽ Результат снова ожидаемый, но создан новый `Vector`.

Если в классе не реализованы операторы «на месте», перечисленные в табл. 16.1, то операторы составного присваивания – не более чем синтаксический сахар: `a += b` вычисляется точно так же, как `a = a + b`. Это ожидаемое поведение для неизменяемых типов, и если добавить метод `__add__`, то `+=` будет работать безо всякого дополнительного кода.

Однако если все-таки реализовать метод оператора «на месте», например `__iadd__`, то он и будет вызван для вычисления выражения `a += b`. Как следует из названия, такие операторы изменяют сам левый операнд, а не создают новый объект-результат.

Специальные методы, вычисляемые на месте, никогда не следует реализовывать для неизменяемых типов и, в частности, для нашего класса `Vector`. Это, в общем-то, очевидно, но лишний раз подчеркнуть не помешает.

Чтобы продемонстрировать код оператора «на месте», мы расширим класс `BingoCage` из примера 13.9, реализовав в нем методы `__add__` и `__iadd__`.

Назовем подкласс `AddableBingoCage`. В примере 16.17 показано, какого поведения мы ожидаем от оператора `+`.

Пример 16.17. Оператор `+` создает новый экземпляр `AddableBingoCage`

```
>>> vowels = 'AEIOU'
>>> globe = AddableBingoCage(vowels) ❶
>>> globe.inspect()
('A', 'E', 'I', 'O', 'U')
>>> globe.pick() in vowels ❷
True
>>> len(globe.inspect()) ❸
4
>>> globe2 = AddableBingoCage('XYZ') ❹
>>> globe3 = globe + globe2
>>> len(globe3.inspect()) ❺
7
>>> void = globe + [10, 20] ❻
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for +: 'AddableBingoCage' and 'list'
```

❶ Создать объект `globe` с пятью элементами (все гласные буквы).
❷ Извлечь один элемент и проверить, что это гласная.
❸ Убедиться, что количество элементов в `globe` уменьшилось до четырех.

❹ Создать второй экземпляр с тремя элементами.
❺ Создать третий экземпляр, складывая первые два. В этом экземпляре семь элементов.
❻ Попытка сложить `AddableBingoCage` со списком приводит к исключению `TypeError`. Такое сообщение интерпретатор Python генерирует, когда наш метод `__add__` возвращает `NotImplemented`.

Объект `AddableBingoCage` изменяемый, и в примере 16.18 показано, как он ведет себя после добавления метода `__iadd__`.

Пример 16.18. Существующий объект `AddableBingoCage` можно модифицировать с помощью оператора `+=` (продолжение примера 16.17)

```
>>> globe_orig = globe ❶
>>> len(globe.inspect()) ❷
4
>>> globe += globe2 ❸
>>> len(globe.inspect())
7
>>> globe += ['M', 'N'] ❹
>>> len(globe.inspect())
9
>>> globe is globe_orig ❺
True
>>> globe += 1 ❻
Traceback (most recent call last):
  ...
TypeError: right operand in += must be 'Tombola' or an iterable
```

❶ Создать псевдоним, чтобы можно было проверить идентификатор объекта позже.
❷ Здесь `globe` содержит четыре элемента.
❸ Объект `AddableBingoCage` может получать элементы от другого объекта того же класса.
❹ Правый операнд `+=` может быть любым итерируемым объектом.
❺ В этом примере `globe` все время ссылается на объект `globe_orig`.
❻ Попытка сложить `AddableBingoCage` с неитерируемым объектом приводит к исключению `TypeError` с надлежащим сообщением.

Отметим, что оператор `+=` либеральнее, чем `+`, относится ко второму операнду. В случае `+` мы хотели, чтобы оба операнда имели одинаковый тип (в данном случае `AddableBingoCage`), потому что иначе было бы не понятно, какой тип должен иметь результат. Для `+=` ситуация проще: левый объект обновляется на месте, поэтому тип результата не вызывает сомнений.

Различия в поведении операторов `+` и `+=` я обосновал, наблюдая за работой встроенного типа `list`. Запись `my_list + x` позволяет конкатенировать только один список с другим, но если написать `my_list += x`, то в правой части может стоять любой итерируемый объект `x`. Это согласуется с поведением метода `list.extend()`: он принимает произвольный итерируемый аргумент.

Поняв, чего мы хотим от класса `AddableBingoCage`, рассмотрим его реализацию (пример 16.19). Напомним, что `BingoCage` из примера 13.9 – конкретный подкласс ABC `Tombola` из примера 13.7.

Пример 16.19. bingoaddable.py: класс `AddableBingoCage` расширяет `BingoCage`, добавляя поддержку операторов `+` и `+=`

```
from tombola import Tombola
from bingo import BingoCage

class AddableBingoCage(BingoCage): ❶

    def __add__(self, other):
        if isinstance(other, Tombola): ❷
            return AddableBingoCage(self.inspect() + other.inspect())
        else:
            return NotImplemented

    def __iadd__(self, other):
        if isinstance(other, Tombola):
            other_iterable = other.inspect() ❸
        else:
            try:
                other_iterable = iter(other) ❹
            except TypeError: ❺
                msg = ('right operand in += must be '
                       "'Tombola' or an iterable")
                raise TypeError(msg)
        self.load(other_iterable) ❻
        return self ❼
```

❶ `AddableBingoCage` расширяет `BingoCage`.
❷ Наш метод `__add__` работает, только когда вторым операндом является объект класса `Tombola`.
❸ В `__iadd__` получить элементы из `other`, если это экземпляр `Tombola`.
❹ В противном случае попытаться получить итератор для `other`[1].
❺ В случае ошибки возбудить исключение, объясняя пользователю, что делать. По возможности сообщения об ошибках должны содержать ясное указание, как решить проблему.
❻ Если мы дошли до этого места, то можем загрузить объект `other_iterable` в `self`.
❼ Очень важно: специальные методы операторов составного присваивания должны возвращать `self`. Именно этого ожидает пользователь.

Резюмировать идею операторов «на месте» можно, сравнив предложения `return`, которые возвращают результаты в методах `__add__` и `__iadd__` из примера 16.19:

`__add__`

Результат порождается путем вызова конструктора `AddableBingoCage` для создания нового экземпляра.

[1] Встроенная функция `iter` рассматривается в следующей главе. Здесь я мог бы написать `tuple(other)`, и это работало бы, но ценой построения нового кортежа, хотя методу `.load(…)` нужно только обойти свой аргумент.

`__iadd__`

Результат порождается путем возврата `self` после модификации.

И последнее замечание к примеру 16.19: в классе `AddableBingoCage` я сознательно не стал реализовывать метод `__radd__`, т. к. в нем нет необходимости. Прямой метод `__add__` работает, только когда правый операнд имеет тот же тип, что левый, поэтому если Python попытается вычислить `a + b`, где `a` принадлежит типу `AddableBingoCage`, а `b` – нет, то получит в ответ `NotImplemented` – быть может, сумеет справиться класс объекта `b`. Но при вычислении выражения `b + a`, когда `b` не принадлежит типу `AddableBingoCage` и возвращает `NotImplemented`, лучше позволить интерпретатору сдаться и возбудить исключение `TypeError`, поскольку мы не умеем обрабатывать `b`.

В общем случае, если прямой инфиксный оператор (например, `__mul__`) предназначен для работы только с операндами того же типа, что `self`, бесполезно реализовывать соответствующий инверсный метод (например, `__rmul__`), потому что он, по определению, вызывается, только когда второй операнд имеет другой тип.

На этом мы завершаем рассмотрение перегрузки операторов в Python.

Резюме

Мы начали эту главу с обзора ограничений, который Python налагает на перегрузку операторов: запрещается перегружать операторы встроенных типов, запрещается создавать новые операторы и перегружать операторы `is`, `and`, `or` и `not`.

Потом мы занялись унарными операторами и реализовали методы `__neg__` и `__pos__`. Далее перешли к инфиксным операторам, начав с `+` и поддерживающего его метода `__add__`. Мы видели, что унарные и инфиксные операторы должны возвращать новый объект в качестве результата и не должны изменять свои операнды. Чтобы поддержать операции с разными типами, мы возвращаем специальное значение `NotImplemented` – не исключение, – давая интерпретатору возможность попробовать еще раз: поменять операнды местами и вызвать специальный инверсный метод, соответствующий тому же оператору (например, `__radd__`). Алгоритм работы с инфиксными операторами в Python показан на рис. 16.1.

Раз мы можем производить операции над объектами разных типов, то должны уметь определять, что нам подсунули операнд, который мы не способны обработать. Мы применяли для этого два способа: либо в духе утиной типизации пробовали выполнить операцию и перехватывали возможное исключение `TypeError`, либо – в методе `__mul__` – явно проверяли тип с помощью `isinstance`. У обоих подходов есть свои плюсы и минусы: утиная типизация обладает большей гибкостью, а явная проверка типов дает более предсказуемый результат.

В общем случае библиотекам лучше использовать утиную типизацию, держа дверь открытой для любых объектов, поддерживающих необходимые операции, вне зависимости от их типа. Однако алгоритм диспетчеризации операторов в Python может выдавать вводящие в заблуждение сообщения об ошибках или возвращать неожиданные результаты, если сочетается с утиной типизацией. По этой причине при написании специальных методов для перегрузки

операторов часто бывает полезно сравнивать тип с ABC посредством `isinstance`. Эта техника, которую Алекс Мартелли окрестил гусиной типизацией (см. раздел «Гусиная типизация» главы 13), предлагает разумный компромисс между гибкостью и безопасностью, поскольку существующие или будущие пользовательские типы можно объявить как настоящие или виртуальные подклассы ABC. Кроме того, если ABC реализует метод `__subclasshook__`, то объект пройдет проверку `isinstance` на предмет сравнения с этим ABC, если предоставит требуемые методы – ни наследования, ни регистрации при этом не требуется.

Далее мы обсудили операторы сравнения. Мы реализовали оператор `==` с помощью метода `__eq__` и выяснили, что Python предоставляет удобную реализацию оператора `!=` в форме метода `__ne__`, унаследованного от базового класса `object`. Эти операторы, а также `>`, `<`, `>=` и `<=` Python вычисляет несколько иначе, применяя различную логику для выбора инверсного метода и специальный запасной вариант для операторов `==` и `!=` – в этом случае исключение никогда не возбуждается, потому что интерпретатор в качестве последнего средства сравнивает идентификаторы объектов.

Последний раздел был посвящен операторам составного присваивания. Мы видели, что Python по умолчанию рассматривает их как комбинацию обычного оператора и присваивания, то есть `a += b` вычисляется точно так же, как `a = a + b`. При этом всегда создается новый объект, так что оператор одинаково хорошо работает для изменяемых и неизменяемых типов. Но для изменяемых типов мы можем реализовать специальные методы, вычисляемые на месте, например `__iadd__` для оператора `+=`, и модифицировать значение левого операнда. Чтобы продемонстрировать эту возможность, мы расстались с неизменяемым классом `Vector` и занялись реализацией подкласса `BingoCage`, поддерживающего оператор `+=` для добавления элементов в случайный пул – по аналогии с тем, как встроенный тип `list` поддерживает оператор `+=`, являющийся сокращенной записью метода `list.extend()`. Попутно мы обсудили, почему оператор `+` ведет себя более разборчиво, чем `+=`, в том, что касается допустимых типов операндов. Для типов последовательностей `+` обычно требуется, чтобы оба операнда имели одинаковый тип, тогда как `+=` зачастую принимает произвольный итерируемый объект в качестве правого операнда.

Дополнительная литература

Гвидо ван Россум выступил в защиту перегрузки операторов в статье «Why operators are useful» (*https://neopythonic.blogspot.com/2019/03/why-operators-are-useful.html*). Трей Ханнер в своем блоге разместил пост «Tuple ordering and deep comparisons in Python» (*https://treyhunner.com/2019/03/python-deep-comparisons-and-code-readability/*), доказывая, что операторы сравнения в Python являются более гибкими и мощными, чем полагают программисты, знакомые с другими языками.

Перегрузка операторов – одна из областей программирования на Python, где проверки с помощью `isinstance` – обычное дело. Для таких проверок рекомендуется гусиная типизация, рассмотренная в одноименном разделе главы 13. Если вы ее пропустили, прочитайте сейчас.

Основным источником информации о специальных методах операторов является глава «Модель данных» (*https://docs.python.org/3/reference/datamodel.*

html) справочного руководства. Еще одна относящаяся к теме часть документации – раздел 9.1.2.2 «Реализация арифметических операций» (*https://docs.python.org/3/library/numbers.html#implementing-the-arithmetic-operations*) в описании модуля `numbers` стандартной библиотеки Python.

Изобретательный пример перегрузки операторов имеется в пакете `pathlib` (*https://docs.python.org/3/library/pathlib.html*), добавленном в версии Python 3.4. Класс `Path` перегружает оператор `/` с целью построения путей в файловой системе из строк. Как это делается, показывает пример, взятый из документации:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
```

Еще один пример перегрузки операторов, не относящийся к арифметике, дает библиотека Scapy (*https://pypi.org/project/scapy/*), предназначенная для «отправки, прослушивания, анализа и подделки сетевых пакетов». В Scapy оператор `/` строит пакеты, собирая поля из различных сетевых уровней. Детали см. в разделе документации «Stacking layers» (*https://scapy.readthedocs.io/en/latest/usage.html# stacking-layers*).

Если вы собираетесь реализовать операторы сравнения, изучите функцию `functools.total_ordering`. Это декоратор класса, который автоматически генерирует недостающие методы для всех операторов сравнения в любом классе, где есть хотя бы два из них. См. документацию по модулю `functools` (*https://docs.python.org/3/library/functools.html#functools.total_ordering*).

Если вам интересно узнать о диспетчеризации операторных методов в языках с динамической типизацией, почитайте две основополагающие работы: Дэн Инголлс (один из разработчиков Smalltalk) «A Simple Technique for Handling Multiple Polymorphism» (*https://dl.acm.org/doi/10.1145/960112.28732*) и Курт Дж. Гебель, Ральф Джонсон «Arithmetic and Double Dispatching in Smalltalk-80» (*https://www.researchgate.net/publication/239578755_Arithmetic_and_double-_dispatching_in_smalltalk-80*) (Джонсон впоследствии стал знаменит как один из авторов книги «Паттерны проектирования»). В обеих статьях глубоко проработан вопрос о полиморфизме в языках с динамической типизацией, к каковым относятся Smalltalk, Python и Ruby. В Python для обработки операторов не применяется двойная диспетчеризация, описанная в этих статьях. Используемый в Python алгоритм на основе прямого и инверсного операторов проще поддержать в пользовательских классах, чем двойную диспетчеризацию, но он требует специального внимания со стороны интерпретатора. Напротив, классическая двойная диспетчеризация – это общая техника, применимая как в Python, так и в любом другом объектно-ориентированном языке, – и не только в контексте инфиксных операторов. На самом деле Инголлс, Гебель и Джонсон иллюстрируют ее на самых разных примерах.

Статья «The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling» (*http://www.gotw.ca/publications/c_family_interview.htm*), из которой взят эпиграф к этой главе, а также две цитаты во врезке «Поговорим», была опубликована в журналах «Java Report», 5 (7), июль 2000, и «C++ Report», 12 (7), июль-август 2000. Это очень увлекательное чтение для всех, кто интересуется проектированием языков программирования.

Поговорим

Перегрузка операторов: за и против

Джеймс Гослинг, процитированный в эпиграфе к этой главе, сознательно решил не включать перегрузку операторов в язык Java. В том же интервью («The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling» – *http://www.gotw.ca/publications/c_family_interview.htm*) он говорит:

> Наверное, от 20 до 30 процентов людей считают перегрузку операторов порождением дьявола; кто-то делал с помощью перегрузки операторов нечто такое, что напрочь выносит мозг, поскольку использование + для вставки в список способно привести в полное замешательство. Проблема проистекает главным образом из того, что есть всего пяток операторов, перегружать которые имеет смысл, и тысячи, а то и миллионы операторов, которые программисты хотели бы определить. Поэтому приходится выбирать, и зачастую выбор входит в противоречие с интуицией.

Гвидо ван Россум выбрал средний путь: он не оставил пользователям открытую дверь для определения новых операторов, например `<=>` или `:-)`, чем предотвратил возведение Вавилонской башни нестандартных операторов и переусложнение синтаксического анализатора. Python также не позволяет перегружать операторы встроенных типов, и это ограничение тоже способствует удобочитаемости и обеспечивает предсказуемую производительность.

Гослинг продолжает:

> Из всего сообщества примерно 10 процентов используют перегрузку операторов надлежащим образом и относятся к ней ответственно, им она действительно необходима. Это почти исключительно люди, занимающиеся численными расчетами, где нотация обязательно должна быть интуитивно очевидной; возможность написать «a + b», где a и b – комплексные числа, матрицы или еще что-то в этом роде, действительно полезна.

Разумеется, и у решения запретить перегрузку операторов в языке есть свои плюсы. Я встречал мнение, что C лучше, чем C++ для системного программирования, потому что из-за перегрузки операторов в C++ дорогостоящие операции могут показаться тривиальными. В двух современных успешных языках, которые компилируются в двоичные исполняемые файлы, приняты противоположные решения по этому поводу: в Go нет перегрузки операторов, а в Rust есть (*https://doc.rust-lang.org/std/ops/ index.html*).

Но при разумном использовании перегруженные операторы упрощают чтение и написание кода. В современных высокоуровневых языках эта возможность очень полезна.

Беглый взгляд на отложенные вычисления

Внимательно присмотревшись к обратной трассировке в примере 16.9, вы заметите следы *отложенного*, или *ленивого*, вычисления генераторных выражений. В примере 16.20 показана та же трассировка, но с выносками.

Пример 16.20. Повторение примера 16.9

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs) ❶
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components) ❷
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs) ❸
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

❶ Конструктору `Vector` передается генераторное выражение в аргументе `components`. Пока никаких проблем.

❷ Генераторное выражение `components` передается конструктору массива `array`. Внутри конструктора Python пытается обойти генераторное выражение, что приводит к вычислению первого элемента `a + b`. Именно в этот момент происходит исключение `TypeError`.

❸ Исключение распространяется в вызов конструктора `Vector`, и там печатается сообщение о нем.

Отсюда видно, что генераторное выражение вычисляется в последний момент, а не там, где оно определено в исходном коде.

С другой стороны, если бы конструктор `Vector` был вызван как `Vector([a + b for a, b in pairs])`, то исключение произошло бы прямо здесь, поскольку списковое включение попыталось бы построить список для передачи в качестве аргумента конструктору `Vector()`. До метода `Vector.__init__` дело вообще не дошло бы.

Мы будем детально рассматривать генераторные выражения в главе 17, но я не хотел упустить случай продемонстрировать их ленивую природу.

Часть IV

Поток управления

Глава 17

Итераторы, генераторы и классические сопрограммы

> Видя в своих программах повторяющиеся структуры, я расцениваю их как знак беды. Форма программы должна отражать задачу, которую она призвана решить, и только ее. Любые другие регулярности в коде означают, по крайней мере для меня, что я использую недостаточно выразительные абстракции – зачастую из-за того, что вручную расширяю макросы, которые должен был бы написать.
>
> – Пол Грэхем, знаток Lisp и венчурный инвестор[1]

Итерирование – одна из важнейших операций обработки данных. А если просматривается набор данных, не помещающийся целиком в память, то нужен способ выполнять ее *лениво*, т. е. по одному элементу и по запросу. Именно это и делает *Итератор*. В этой главе мы покажем, что паттерн *Итератор* встроен в язык Python, поэтому реализовывать его вручную вам никогда не придется.

Любая стандартная коллекция в Python является *итерируемым объектом*, т. е. предоставляет *итератор*, который используется для поддержки следующих операций:

- циклов `for`;
- списковых, словарных и множественных включений;
- распаковки операций присваивания;
- конструирования экземпляров коллекций.

В этой главе рассматриваются следующие темы:

- как встроенная функция `iter(…)` используется интерпретатором для обработки итерируемых объектов;
- как реализовать классический паттерн Итератор в Python;
- как можно заменить классический Итератор генераторной функцией или генераторным выражением;
- подробное построчное описание работы генераторной функции;
- использование генераторных функций общего назначения в стандартной библиотеке;
- использование предложения `yield from` для комбинирования генераторов;
- почему генераторы и классические сопрограммы, несмотря на внешнюю схожесть, по существу сильно различаются и не должны использоваться совместно.

[1] Из статьи в блоге «Revenge of the Nerds» (*http://www.paulgraham.com/icad.html*).

Что нового в этой главе

Раздел «Субгенераторы, содержащие yield from» разросся с одной до шести страниц. Теперь он включает простые эксперименты, демонстрирующие поведение генераторов с `yield from`, и пошаговую разработку примера – обхода древовидной структуры данных.

В новых разделах объясняется применение аннотаций для типов `Iterable`, `Iterator` и `Generator`.

Последний раздел этой главы «Классические сопрограммы» сжался до 9-страничного введения в тему, которая в первом издании занимала целую главу объемом 40 страниц. Я переработал эту главу и перенес ее в виде статьи на сопроводительный сайт (*https://www.fluentpython.com/extra/classic-coroutines/*), поскольку, с одной стороны, она была самой трудной для читателей, а с другой – ее предмет стал менее актуальным, после того как в Python 3.5 появились платформенные сопрограммы, которые мы будем изучать в главе 21.

Начнем с вопроса о том, как встроенная функция `iter(…)` делает последовательность итерируемой.

Последовательность слов

Исследование итерируемых объектов мы начнем с реализации класса `Sentence`: его конструктору передается текстовая строка, после чего ее можно перебирать слово за словом. В первой версии мы реализуем протокол последовательности, итерируемость будет достигнута за счет того, что все последовательности – итерируемые объекты, но теперь мы точно узнаем, почему.

В примере 17.1 приведен класс `Sentence`, который умеет извлекать из текста слово с заданным индексом.

Пример 17.1. sentence.py: объект `Sentence` как последовательность слов

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text) ❶

    def __getitem__(self, index):
        return self.words[index] ❷

    def __len__(self): ❸
        return len(self.words)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text) ❹
```

❶ `.findall` возвращает список всех непересекающихся подстрок, соответствующих регулярному выражению.

❷ `self.words` содержит результат `.findall`, поэтому мы просто возвращаем слово с заданным индексом.

❸ Чтобы выполнить требования протокола последовательности, мы реализуем метод `__len__`, но для получения итерируемого объекта он не нужен.
❹ Служебная функция `reprlib.repr` генерирует сокращенные строковые представления структур данных, которые могут быть очень велики[1].

По умолчанию `reprlib.repr` ограничивает сгенерированную строку 30 символами. В примере 17.2 показано, как используется класс `Sentence`.

Пример 17.2. Итерирование объекта `Sentence`

```
>>> s = Sentence('"The time has come," the Walrus said,') ❶
>>> s
Sentence('"The time ha... Walrus said,') ❷
>>> for word in s: ❸
...     print(word)
The
time
has
come
the
Walrus
said
>>> list(s) ❹
['The', 'time', 'has', 'come', 'the', 'Walrus', 'said']
```

❶ По строке создается предложение – объект класса `Sentence`.
❷ Обратите внимание на результат `__repr__` – строку, содержащую многоточие, которая была сгенерирована функцией `reprlib.repr`.
❸ Объекты `Sentence` являются итерируемыми, скоро мы в этом убедимся.
❹ Будучи итерируемыми, объекты `Sentence` могут быть использованы для конструирования списков и других итерируемых типов.

Далее мы разработаем другие классы `Sentence`, которые будут успешно проходить тесты из примера 17.2. Но реализация из примера 17.1 отличается от всех остальных тем, что является также последовательностью, а значит, допускает доступ к слову по индексу.

```
>>> s[0]
'The'
>>> s[5]
'Walrus'
>>> s[-1]
'said'
```

Любой программирующий на Python знает, что последовательности – итерируемые объекты. Разберемся, почему это так.

Почему последовательности итерируемы: функция iter

Всякий раз как интерпретатору нужно обойти объект `x`, он автоматически вызывает функцию `iter(x)`.

[1] Впервые мы встретились с ней в разделе «Vector, попытка № 1: совместимость с Vector2d» главы 12.

Встроенная функция `iter` выполняет следующие действия.

1. Смотрит, реализует ли объект метод `__iter__`, и, если да, вызывает его, чтобы получить итератор.
2. Если метод `__iter__` не реализован, но реализован метод `__getitem__`, то Python создает итератор, который пытается извлекать элементы по порядку, начиная с индекса 0.
3. Если и это не получается, то возбуждается исключение – обычно с сообщением `'C' object is not iterable`, где `C` – класс объекта.

Именно поэтому любая последовательность в Python является итерируемой: все они реализуют метод `__getitem__`. На самом деле стандартные последовательности реализуют и метод `__iter__`, и ваши должны поступать так же, поскольку специальная обработка метода `__getitem__` оставлена только ради обратной совместимости и может быть исключена в будущем – хотя она не объявлена нерекомендуемой в Python 3.10 и сомнительно, что будет когда-нибудь удалена.

В разделе «Python в поисках следов последовательностей» главы 13 отмечалось, что это крайняя форма утиной типизации: объект считается итерируемым не только, когда он реализует специальный метод `__iter__`, но и когда реализует метод `__getitem__`. Взгляните:

```
>>> class Spam:
...     def __getitem__(self, i):
...         print('->', i)
...         raise IndexError()
...
>>> spam_can = Spam()
>>> iter(spam_can)
<iterator object at 0x10a878f70>
>>> list(spam_can)
-> 0
[]
>>> from collections import abc
>>> isinstance(spam_can, abc.Iterable)
False
```

Если класс предоставляет метод `__getitem__`, то встроенная функция `iter()` принимает экземпляр этого класса в качестве итерируемого объекта и строит по нему итератор. Механизм итерирования Python будет вызывать `__getitem__` с индексами, начинающимися с 0, и воспринимать исключение `IndexError` как сигнал о том, что элементы кончились.

Заметим, что хотя `spam_can` – итерируемый объект (его метод `__getitem__` мог бы поставлять элементы), он не распознается как таковой функцией `isinstance`, когда она сравнивает его с `abc.Iterable`.

Если подходить с точки зрения гусиной типизации, то определение итерируемого объекта становится более простым, но не таким гибким: объект считается итерируемым, если реализует метод `__iter__`. Не требуется ни наследования, ни регистрации, потому что класс `abc.Iterable` реализует метод `__subclasshook__` (см. раздел «ABC и структурная типизация» главы 13). Продемонстрируем это:

```
>>> class GooseSpam:
...     def __iter__(self):
...     pass
```

```
...
>>> from collections import abc
>>> issubclass(GooseSpam, abc.Iterable)
True
>>> goose_spam_can = GooseSpam()
>>> isinstance(goose_spam_can, abc.Iterable)
True
```

В версии Python 3.10 самый точный способ проверить, является ли объект `x` итерируемым, – вызвать `iter(x)` и перехватить исключение `TypeError`, если оно возникнет. Это надежнее, чем использовать `isinstance(x, abc.Iterable)`, потому что `iter(x)` учитывает также доставшийся в наследство метод `__getitem__`, а ABC `Iterable` этого не делает.

Явно проверять, является ли объект итерируемым, вряд ли стоит, если сразу после проверки вы намереваетесь обойти объект. Ведь если попытаться обойти неитерируемый объект, Python возбудит исключение с недвусмысленным сообщением: `TypeError: 'C' object is not iterable`. Если вы можете сделать что-то более разумное, чем возбуждать `TypeError`, делайте это в блоке `try/except`, а не путем явной проверки. Явная проверка, возможно, имеет смысл, если вы хотите сохранить объект и воспользоваться им для итерирования позже; в таком случае раннее обнаружение ошибки упрощает отладку.

Встроенная функция `iter()` чаще вызывается самим интерпретатором Python, чем вашим кодом. Есть другой, не столь хорошо известный способ ее использования.

Использование iter в сочетании с Callable

Мы можем вызвать `iter()` с двумя аргументами, чтобы создать итератор из функции или вообще любого вызываемого объекта. В таком случае первый аргумент должен быть вызываемым объектом, который будет вызываться многократно (без аргументов) для порождения значений, а второй – *специальным маркером* (*https://en.wikipedia.org/wiki/Sentinel_value*): если вызываемый объект возвращает такое значение, то итератор не отдает его вызывающей стороне, а возбуждает исключение `StopIteration`.

В примере ниже показано, как можно использовать `iter` для бросания шестигранной кости до тех пор, пока не выпадет `1`:

```
>>> def d6():
... return randint(1, 6)
...
>>> d6_iter = iter(d6, 1)
>>> d6_iter
<callable_iterator object at 0x10a245270>
>>> for roll in d6_iter:
...     print(roll)
...
4
3
6
3
```

Заметим, что функция `iter` в этом примере возвращает `callable_iterator`. Цикл `for` может работать очень долго, но никогда не выведет `1`, потому что это значение является специальным маркером. Как всегда бывает с итераторами, объект `d6_iter` после исчерпания становится бесполезным. Чтобы начать с начала, мы должны создать новый итератор, снова вызвав `iter()`.

В документации по `iter` (*https://docs.python.org/3.10/library/functions.html#iter*) имеется следующее объяснение и пример:

> Вторую форму `iter()` можно с пользой применить для построения читателя блоков. Например, вот как можно читать блоки фиксированной длины из двоичного файла базы данных, до тех пор пока не будет достигнут конец файла:

```
from functools import partial

with open('mydata.db', 'rb') as f:
    read64 = partial(f.read, 64)
    for block in iter(read64, b''):
        process_block(block)
```

Для ясности я добавил присваивание переменной `read64`, которого нет в оригинале (*https://docs.python.org/3.10/library/functions.html#iter*). Функция `partial()` необходима, потому что вызываемый объект, переданный `iter()`, не должен иметь аргументов. В примере специальным маркером является пустой объект `bytes`, потому что именно его возвращает `f.read`, когда больше нечего читать.

В следующем разделе мы проясним связь между итерируемыми объектами и итераторами.

Итерируемые объекты и итераторы

Из объяснения в разделе «Почему последовательности итерируемы: функция iter» можно вывести такое определение:

Итерируемый объект

> Любой объект, от которого встроенная функция `iter` может получить итератор. Объекты, которые реализуют метод `__iter__`, возвращающий *итератор*, являются итерируемыми. Последовательности всегда итерируемы, поскольку это объекты, реализующие метод `__getitem__`, который принимает индексы, начинающиеся с нуля.

Важно четко понимать связь между итерируемыми объектами и итераторами: Python получает итераторы от итерируемых объектов.

Ниже приведен простой цикл `for` для обхода строки `str`. Строка `'ABC'` здесь является итерируемым объектом. Мы этого не видим, но за кулисами прячется итератор:

```
>>> s = 'ABC'
>>> for char in s:
...     print(char)
...
ABC
```

Если бы не было предложения `for` и мы должны были бы эмулировать механизм работы `for` вручную с помощью цикла `while`, то пришлось бы написать такой код:

```
>>> s = 'ABC'
>>> it = iter(s) ❶
>>> while True:
...     try:
...         print(next(it)) ❷
...     except StopIteration: ❸
...         del it ❹
...         break ❺
...
A
B
C
```

❶ Получить итератор `it` от итерируемого объекта.
❷ В цикле вызвать метод `next` итератора, чтобы получить следующий элемент.
❸ Итератор возбуждает исключение `StopIteration`, когда элементы кончаются.
❹ Освободить ссылку на `it` – объект итератора уничтожается.
❺ Выйти из цикла.

Исключение `StopIteration` сигнализирует об исчерпании итератора. В циклах `for` и в других контекстах итерирования, например в списковом включении, при распаковке итерируемых объектов и т. д., оно обрабатывается встроенной функцией `iter()`.

В стандартном интерфейсе итератора есть два метода:

`__next__`
: Возвращает следующий доступный элемент и возбуждает исключение `StopIteration`, когда элементов не осталось.

`__iter__`
: Возвращает `self`; это позволяет использовать итератор там, где ожидается итерируемый объект, например в цикле `for`.

Этот интерфейс формализован в абстрактном базовом классе `collections.abc.Iterator`, где определен абстрактный метод `__next__`, и в его подклассе `Iterable`, где определен абстрактный метод `__iter__` (см. рис. 17.1).

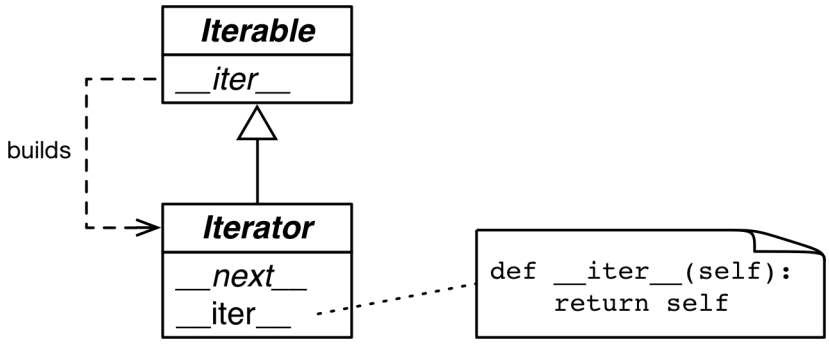


Рис. 17.1. Абстрактные классы `Iterable` и `Iterator`. Курсивом набраны имена абстрактных методов. Конкретный метод `Iterable.iter` должен возвращать новый экземпляр `Iterator`. Конкретный `Iterator` должен реализовывать метод `next`. Метод `Iterator.iter` просто возвращает ссылку на себя

Исходный код класса `collections.abc.Iterator` приведен в примере 17.3.

Пример 17.3. Класс `abc.Iterator`; код взят из файла *Lib/_collections_abc.py* (*https://github.com/python/cpython/blob/b1930bf75f276cd7ca08c4455298128d89adf7d1/Lib/_collections_abc.py#L271*)

```
class Iterator(Iterable):

    __slots__ = ()

    @abstractmethod
    def __next__(self):
        'Return the next item from the iterator. When exhausted,
raise StopIteration'
        raise StopIteration

    def __iter__(self):
        return self

    @classmethod
    def __subclasshook__(cls, C): ❶
        if cls is Iterator:
            return _check_methods(C, '__iter__', '__next__') ❷
        return NotImplemented
```

❶ `__subclasshook__` поддерживает структурные проверки типов с помощью функций `isinstance` и `issubclass`. Мы видели это в разделе «ABC и структурная типизация» главы 13.

❷ Метод `_check_methods` обходит `__mro__` класса и проверяет, реализованы ли указанные методы в его базовых классах. Он определен в том же модуле *Lib/_collections_abc.py*. Если методы реализованы, то класс `C` будет распознан как виртуальный подкласс `Iterator`. Иными словами, `issubclass(C, Iterable)` вернет `True`.

В Python 3 абстрактный метод класса `Iterator` называется `it.__next__()`, а в Python 2 – `it.next()`. Как обычно, не следует вызывать специальные методы напрямую. Просто пользуйтесь встроенной функцией `next(it)`: она сделает все правильно – и в Python 2, и в Python 3.

В исходном файле *Lib/types.py* (*https://hg.python.org/cpython/file/3.4/Lib/types.py*) для версии Python 3.9 есть такой комментарий:

```
# Итераторы в Python следует считать не типом, а протоколом. Многие
# встроенные типы (их число постоянно изменяется) реализуют *какой-то*
# вид итератора. Не проверяйте его тип! Используйте вместо этого
# функцию hasattr для проверки наличия атрибутов «__iter__» и «__next__».
```

На самом деле именно это и делает метод `__subclasshook__` абстрактного класса `abc.Iterator`.

Принимая во внимание рекомендацию из файла *Lib/types.py* и логику, реализованную в файле *Lib/_collections_abc.py*, согласимся, что лучший способ узнать, является ли объект `x` итератором, – вызвать функцию `isinstance(x, abc.Iterator)`. Благодаря методу `Iterator.__subclasshook__` эта проверка работает даже тогда, когда класс `x` не является ни настоящим, ни виртуальным подклассом `Iterator`.

Возвращаясь к классу `Sentence` из примера 17.1, мы можем в интерактивной оболочке посмотреть, как итератор строится функцией `iter(…)` и производит обход с помощью `next(…)`:

```
>>> s3 = Sentence('Life of Brian') ❶
>>> it = iter(s3) ❷
>>> it # doctest: +ELLIPSIS
<iterator object at 0x...>
>>> next(it) ❸
'Life'
>>> next(it)
'of'
>>> next(it)
'Brian'
>>> next(it) ❹
Traceback (most recent call last):
  ...
StopIteration
>>> list(it) ❺
[]
>>> list(iter(s3)) ❻
['Life', 'of', 'Brian']
```

❶ Создать предложение `s3`, содержащее три слова.
❷ Получить от `s3` итератор.
❸ `next(it)` возвращает следующее слово.
❹ Больше слов нет, поэтому итератор возбуждает исключение `StopIteration`.
❺ После исчерпания итератор бесполезен.
❻ Чтобы еще раз обойти предложение, нужно создать новый итератор.

Поскольку от итератора требуются только методы `__next__` и `__iter__`, не существует другого способа узнать, остались ли еще элементы, как только вызвать `next()` и перехватить исключение `StopInteration`. И «сбросить» итератор тоже невозможно. Чтобы начать обход сначала, нужно вызвать функцию `iter()` для итерируемого объекта и получить от нее новый итератор. Вызов `iter()` для самого итератора не поможет, поскольку, как уже упоминалось, метод `Iterator.__iter__` возвращает `self`, так что таким способом исчерпанный итератор не восстановить.

Этот минимальный интерфейс разумен, потому что на практике не все итераторы допускают сброс. Например, если итератор читает пакеты из сети, то перемотать его в начало невозможно[1].

Первая версия класса `Sentence` из примера 17.1 была итерируемой вследствие специальной обработки последовательностей встроенной функцией `iter()`. Теперь мы напишем вариант `Sentence`, который реализует метод `__iter__` для возврата итераторов.

Классы Sentence с методом __iter__

Последующие варианты `Sentence` реализуют стандартный протокол итерируемого объекта, сначала путем реализации паттерна проектирования Итератор, а затем с помощью генераторных функций.

[1] Спасибо Леонардо Рохаэлю за этот красивый пример.

Класс Sentence, попытка № 2: классический итератор

Следующая версия класса `Sentence` строится согласно классическому паттерну проектирования Итератор, который описан в книге «Банды четырех». Отметим, что это не идиоматический код на Python, что станет предельно понятно, когда мы займемся его рефакторингом. Но он проясняет связь между итерируемой коллекцией и объектом-итератором.

Показанная в примере 17.4 реализация класса `Sentence` является итерируемой, потому что реализует специальный метод `__iter__`, который конструирует и возвращает объект `SentenceIterator`. Именно так работает паттерн проектирования Итератор, описанный в книге «Паттерны проектирования».

Пример 17.4. sentence_iter.py: класс `Sentence`, реализованный с помощью паттерна Итератор

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self): ❶
        return SentenceIterator(self.words) ❷

class SentenceIterator:

    def __init__(self, words):
        self.words = words ❸
        self.index = 0 ❹

    def __next__(self):
        try:
            word = self.words[self.index] ❺
        except IndexError:
            raise StopIteration() ❻
        self.index += 1 ❼
        return word ❽

    def __iter__(self): ❾
        return self
```

❶ Метод `__iter__` – единственное дополнение к предыдущей реализации `Sentence`. В этой версии нет метода `__getitem__`, тем самым мы хотим доказать, что класс является итерируемым, потому что реализует `__iter__`.

❷ `__iter__` выполняет требования протокола итерируемого объекта – создает и возвращает итератор.

❸ `SentenceIterator` хранит ссылку на список слов.
❹ `self.index` используется для определения следующего слова.
❺ Получить слово с индексом `self.index`.
❻ Если слова с индексом `self.index` не существует, возбудить исключение `StopIteration`.
❼ Увеличить `self.index`.
❽ Вернуть слово.
❾ Реализовать метод `self.__iter__`.

Код из примера 17.4 проходит тесты из примера 17.2.

Отметим, что этот пример работал бы и без реализации метода `__iter__` в классе `SentenceIterator`, но лучше все делать правильно: предполагается, что итератор реализует оба метода, `__next__` и `__iter__`, и если мы так сделаем, то наш итератор пройдет проверку `issubclass(SentenceInterator, abc.Iterator)`. Если бы мы унаследовали `SentenceIterator` от `abc.Iterator`, то получили бы и конкретный метод `abc.Iterator.__iter__`.

Что-то многовато работы (по крайней мере, для нас, испорченных программистов на Python). Обратите внимание, что большая часть кода `SentenceIterator` занимается управлением внутренним состоянием итератора. Вскоре мы увидим, как сократить эту часть. Но сначала небольшое отступление, в котором мы опишем один соблазнительный способ срезать угол, который на самом деле никуда не годится.

Не делайте итерируемый объект итератором для самого себя

Типичный источник ошибок при создании итерируемых объектов и итераторов – путаница понятий. Поясним: у итерируемого объекта есть метод `__iter__`, который при каждом обращении создает новый итератор. Итератор реализует метод `__next__`, который возвращает элементы один за другим, и метод `__iter__`, который возвращает `self`.

Следовательно, итератор является итерируемым объектом, но итерируемый объект не является итератором.

Возникает соблазн реализовать в классе `Sentence` метод `__next__` в дополнение к `__iter__` и тем самым сделать экземпляр `Sentence` одновременно итерируемым объектом и итератором над самим собой. Но это редко бывает удачной идеей. Типичный антипаттерн, по словам Алекса Мартелли, у которого огромный опыт рецензирования кода на Python в Google.

В разделе «Применимость» главы о паттерне Итератор в книге «Банды четырех» написано:

> Используйте паттерн Итератор:
>
> - для доступа к содержимому агрегированных объектов без раскрытия их внутреннего представления;
> - для поддержки нескольких активных обходов одного и того же агрегированного объекта;
> - для предоставления единообразного интерфейса с целью обхода различных агрегированных структур (то есть для поддержки полиморфной итерации).

Чтобы «поддержать несколько активных обходов», необходимо иметь возможность получить несколько независимых итераторов от одного итерируе-

мого объекта, причем каждый итератор должен хранить собственное внутреннее состояние, поэтому для правильной реализации паттерна нужно всякий раз обращаться к функции `iter(my_iterable)` за новым независимым итератором. Вот почему нам был необходим класс `SentenceIterator`.

Теперь, продемонстрировав реализацию классического паттерна Итератор, мы можем отложить ее в сторонку. В Python включено ключевое слово `yield` из созданного Барбарой Лисков языка CLU, поэтому нам не нужно «вручную генерировать» код для реализации итераторов.

В следующем разделе представлена идиоматическая реализация класса `Sentence`.

Класс Sentence, попытка № 3: генераторная функция

Реализация той же функциональности в духе Python основана на использовании генераторной функции для замены класса `SequenceIterator`. Объяснение приведено после примера 17.5.

Пример 17.5. sentence_gen.py: реализация класса `Sentence` с помощью генератора

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for word in self.words: ❶
            yield word ❷
        ❸
# это всё! ❹
```

❶ Обойти `self.words`.
❷ Отдать текущее слово.
❸ Явный `return` не нужен; функция может просто «провалиться» и вернет управление автоматически. В любом случае генераторная функция не возбуждает исключения `StopIteration`: когда значений не остается, она просто выходит[1].
❹ Нет нужды в отдельном классе итератора!

[1] Рецензируя этот код, Алекс Мартелли отметил, что тело этого метода можно было бы свести просто к `return iter(self.words)`. Он прав: результатом вызова `__iter__` и в этом случае был бы итератор, как и положено. Но я воспользовался здесь циклом `for` с ключевым словом `yield`, чтобы ввести синтаксис функции-генератора, который будет подробно рассмотрен в следующем разделе. В процессе рецензирования второго издания Леонардо Рохаэль предложил еще одну краткую форму тела `__iter__`: `yield from self.words`. Мы рассмотрим конструкцию `yield from` ниже в этой главе.

Вот и еще одна реализация `Sentence`, которая проходит все тесты из примера 17.2.

В классе `Sentence` в примере 17.4 метод `__iter__` конструировал и возвращал объект `SentenceIterator`. В примере же 17.5 итератор фактически является объектом-генератором, который строится автоматически при вызове метода `__iter__`, потому что `__iter__` здесь – генераторная функция.

Подробное объяснение генераторов приведено ниже.

Как работает генератор

Любая функция в Python, в теле которой встречается ключевое слово `yield`, называется генераторной функцией – при вызове она возвращает объект-генератор. Иными словами, генераторная функция – это фабрика генераторов.

Единственное синтаксическое различие между простой и генераторной функцией – тот факт, что в теле последней встречается ключевое слово `yield`. Некоторые считают, что для генераторных функций следовало бы ввести новое ключевое слово `gen` вместо `def`, но Гвидо не согласен. Его аргументы приведены в документе «PEP 255 – Simple Generators» (*https://www.python.org/dev/peps/pep-0255/*)[1].

В примере 17.6 демонстрируется поведение генераторной функции[2].

Пример 17.6. Генераторная функция, отдающая три числа

```
>>> def gen_123():
...     yield 1 ❶
...     yield 2
...     yield 3
...
>>> gen_123 # doctest: +ELLIPSIS
<function gen_123 at 0x...> ❷
>>> gen_123() # doctest: +ELLIPSIS
<generator object gen_123 at 0x...> ❸
>>> for i in gen_123(): ❹
...     print(i)
1
2
3
>>> g = gen_123() ❺
>>> next(g) ❻
1
>>> next(g)
2
>>> next(g)
3
>>> next(g) ❼
Traceback (most recent call last):
  ...
StopIteration
```

[1] Иногда я включаю префикс или суффикс `gen` в имя генераторной функции, но это не общеупотребительная практика. И разумеется, нельзя этого делать при реализации итерируемого объекта: обязательный специальный метод должен называться `__iter__` – и никак иначе.

[2] Спасибо за пример Дэвиду Квасту.

❶ В теле генераторной функции `yield` часто встречается в цикле, но это необязательно; здесь я просто трижды повторил `yield`.
❷ Приглядевшись, мы увидим, что `gen_123` – объект-функция.
❸ Но при вызове `gen_123()` возвращает объект-генератор.
❹ Объекты-генераторы реализуют интерфейс `Iterator`, поэтому являются также итерируемыми объектами.
❺ Присваиваем новый объект-генератор переменной `g`, чтобы с ним можно было экспериментировать.
❻ Поскольку `g` – итератор, вызов `next(g)` возвращает следующий элемент, порожденный `yield`.
❼ Когда генераторная функция возвращает управление, объект-генератор возбуждает исключение `StopIteration`.

Генераторная функция строит объект-генератор, обертывающий тело функции. При передаче объекта-генератора функции `next()` выполнение продолжается до следующего предложения `yield` в теле функции, а вызов `next()` возвращает значение, порожденное перед приостановкой выполнения функции. Наконец, при возврате из функции обертывающий ее объект-генератор возбуждает исключение `StopIteration` в полном соответствии с протоколом `Iterator`.

Я считаю, что в терминологии, касающейся получения результатов от генератора, лучше соблюдать строгость. Фраза же «генератор *возвращает* значения» вносит путаницу. Значения возвращают функции. Вызов генераторной функции возвращает генератор. Генератор отдает значения. Генератор не «возвращает» значение в обычном смысле слова: предложение `return` в теле генераторной функции приводит к тому, что объект-генератор возбуждает исключение `StopIteration`. Если написать `return x` в генераторе, то вызывающая сторона сможет получить значение `x` из исключения `StopIteration`, но обычно это делается автоматически с помощью синтаксиса `yield from`, как будет показано в разделе «Возврат значения из сопрограммы».

В примере 17.7 во всех подробностях описано взаимодействие между циклом `for` и телом функции.

Пример 17.7. Генераторная функция, печатающая сообщения во время выполнения

```
>>> def gen_AB():
...     print('start')
...     yield 'A'          ❶
...     print('continue')
...     yield 'B'          ❷
...     print('end.')      ❸
...
>>> for c in gen_AB():     ❹
...     print('-->', c)    ❺
...
start     ❻
--> A     ❼
continue  ❽
--> B     ❾
end.      ❿
>>>       ⓫
```

❶ Первый неявный вызов `next()` в цикле `for` в точке ❹ приводит к печати `'start'` и приостановке на первом `yield`, порождающем значение `'A'`.
❷ Второй неявный вызов `next()` в цикле `for` приводит к печати `'continue'` и приостановке на втором `yield`, порождающем значение `'B'`.
❸ Третий вызов `next()` приводит к печати `'end'` и возврату из функции, в результате чего объект-генератор возбуждает исключение `StopIteration`.
❹ Для итерирования цикл `for` выполняет эквивалент предложения `g = iter(gen_AB())`, чтобы получить объект-генератор, а затем на каждой итерации вызывает `next(g)`.
❺ В теле цикла печатается `-->` и значение, полученное от `next(g)`. Но результат этой печати мы увидим только после строки, напечатанной функцией `print` внутри генераторной функции.
❻ Строка `'start'` появляется в результате работы функции `print('start')` в теле генераторной функции.
❼ Предложение `yield 'A'` в теле генераторной функции отдает значение `A`, потребляемое в цикле `for`, где оно присваивается переменной `c` и распечатывается в виде `--> A`.
❽ Итерирование продолжается благодаря второму вызову `next(g)`, продвигающему выполнение генераторной функции от `yield 'A'` к `yield 'B'`. Выводится строка `continue` – результат второго обращения к `print` в теле генераторной функции.
❾ Предложение `yield 'B'` отдает значение `B`, потребляемое в цикле `for`, где оно присваивается переменной `c` и распечатывается в виде `--> B`.
❿ Итерирование продолжается благодаря третьему вызову `next(g)`, продвигающему выполнение в конец генераторной функции. Выводится строка `end` – результат третьего обращения к `print` в теле генераторной функции.
⓫ Когда генераторная функция доходит до конца, объект-генератор возбуждает исключение `StopIteration`. Цикл `for` перехватывает это исключение и нормально завершается.

Надеюсь, теперь понятно, как работает метод `Sentence.__iter__` в примере 14.5: `__iter__` – генераторная функция, которая конструирует объект-генератор, реализующий интерфейс `Iterator`, поэтому класс `SentenceIterator` больше не нужен.

Вторая версия `Sentence` получилась гораздо короче первой, но и она не такая ленивая, какой могла бы быть. В наши дни лень считается хорошим свойством, по крайней мере в языках программирования и API. Ленивая реализация откладывает порождение значений до последней возможности. Это экономит память и иногда позволяет избежать бесполезной работы.

Далее мы напишем ленивый класс `Sentence`.

Ленивые классы Sentence

Последние варианты класса `Sentence` будут ленивыми, на основе ленивой функции из модуля `re`.

Класс Sentence, попытка № 4: ленивый генератор

Интерфейс `Iterator` спроектирован ленивым: вызов `next(my_iterator)` порождает по одному элементу за раз. Противоположностью ленивому вычислению яв-

ляется энергичное (eager) – оба термина применяются в теории языков программирования.

До сих пор наши реализации `Sentence` не были ленивыми, потому что `__init__` энергично строит список всех слов в тексте и связывает его с атрибутом `self.words`. Это влечет за собой обработку всего текста, а список может занять столько же памяти, сколько сам текст (возможно, больше – это зависит от того, сколько в тексте символов, не считающихся частью слова). И большая часть этой работы будет проделана напрасно, если пользователю нужны только первые два слова. Если у вас возникает вопрос «Существует ли ленивый способ сделать это?», ответ чаще всего будет «да».

Функция `re.finditer` – ленивая версия `re.findall`, вместо списка она возвращает генератор, порождающий объекты `re.MatchObject` по запросу. Если соответствий много, то `re.finditer` заметно экономит память. С ее помощью мы напишем третий – ленивый – вариант класса `Sentence`: он читает следующее слово из текста только тогда, когда это необходимо.

Пример 17.8. sentence_gen2.py: реализация класса `Sentence` с помощью генераторной функции, которая вызывает генераторную функцию `re.finditer`

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text ❶

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        for match in RE_WORD.finditer(self.text): ❷
            yield match.group() ❸
```

❶ Хранить список слов не нужно.

❷ `finditer` строит итератор, который обходит все соответствия текста `self.text` регулярному выражению `RE_WORD`, порождая объекты `MatchObject`.

❸ `match.group()` извлекает сопоставленный текст из объекта `MatchObject`.

Генераторы – замечательный способ сократить код, но генераторные выражения еще круче.

Класс Sentence, попытка № 5: генераторное выражение

Простые генераторные функции наподобие той, что использована в предыдущем варианте класса `Sentence` (пример 17.8), можно заменить генераторным выражением. Как списковое включение строит списки, так генераторное выражение строит объекты-генераторы. В примере 17.9 продемонстрировано их поведение.

Пример 17.9. Генераторная функция `gen_AB` используется сначала в списковом включении, а затем в генераторном выражении

```
>>> def gen_AB(): ❶
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()] ❷
start
continue
end.
>>> for i in res1: ❸
...     print('-->', i)
...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB()) ❹
>>> res2
<generator object <genexpr> at 0x10063c240>
>>> for i in res2: ❺
...     print('-->', i)
...
start ❻
--> AAA
continue
--> BBB
end.
```

❶ Та же генераторная функция `gen_AB`, что в примере 17.7.

❷ Списковое включение энергично обходит элементы, порождаемые объектом-генератором, который был создан функцией `gen_AB`: `'A'` и `'B'`. Обратите внимание на печать строк `start`, `continue`, `end`.

❸ В этом цикле `for` мы обходим список `res1`, порожденный списковым включением.

❹ Генераторное выражение возвращает `res2`, объект-генератор. Генератор здесь не потребляется.

❺ Только в цикле `for`, где производится обход `res2`, этот генератор получает элементы от `gen_AB`. На каждой итерации цикла неявно вызывается функция `next(res2)`, которая, в свою очередь, вызывает метод `next()` объекта-генератора, возвращенного `gen_AB`, продвигая генератор до следующего `yield`.

❻ Обратите внимание на то, как строки, напечатанные внутри `gen_AB`, чередуются с теми, что печатаются в самом цикле.

Таким образом, мы можем воспользоваться генераторным выражением, чтобы еще сократить размер класса `Sentence`. См. пример 17.10.

Пример 17.10. sentence_genexp.py: реализация класса `Sentence` с помощью генераторного выражения

```
import re
import reprlib
```

```
RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        return (match.group() for match in RE_WORD.finditer(self.text))
```

От примера 17.8 отличается только метод `__iter__`, который здесь не является генераторной функцией (в нем нет слова `yield`), а использует генераторное выражение для построения генератора, который потом и возвращает. Конечный результат не меняется: код, вызывающий `__iter__`, получает объект-генератор.

Генераторные выражения – не более чем синтаксический сахар: их всегда можно заменить генераторными функциями, но иногда выражения удобнее. Следующий раздел посвящен использованию генераторных выражений.

Генераторные выражения: когда использовать

В реализации класса `Vector` из примера 12.16 я несколько раз пользовался генераторными выражениями. В каждом из методов `__eq__`, `__hash__`, `__abs__`, `angle`, `angles`, `format`, `__add__`, `__mul__` встречается генераторное выражение. Во всех них можно было бы обойтись и списковым включением, но ценой расхода памяти на хранение промежуточных списков.

В примере 17.10 мы видели, что генераторное выражение – синтаксически более короткий способ создать генератор, не определяя и не вызывая функцию. С другой стороны, генераторные функции обладают большей гибкостью: в них можно закодировать сложную логику, включающую несколько предложений, и даже использовать их в качестве *сопрограмм* (см. раздел «Классические сопрограммы»).

В простых случаях генераторное выражение легче воспринимается на взгляд, как показывает пример класса `Vector`.

Я придерживаюсь такого эвристического правила: если генераторное выражение занимает больше двух строк, я предпочитаю генераторную функцию – код получается понятнее.

Синтаксический совет

Когда генераторное выражение передается в качестве единственного аргумента функции или конструктору, нет необходимости указывать одну пару скобок для вызова функции и другую для обрамления генераторного выражения. Хватит и одной пары, как в вызове конструктора `Vector` из метода `__mul__` в примере 12.16 (воспроизведен ниже):

```
    def __mul__(self, scalar):
        if isinstance(scalar, numbers.Real):
            return Vector(n * scalar for n in self)
        else:
            return NotImplemented
```

> Однако если после генераторного выражения есть еще аргументы, то его необходимо заключить в скобки, иначе возникнет исключение `SyntaxError`.

В примерах класса `Sentence` мы видели, как генераторы играют роль классического паттерна Итератор: получают элементы из коллекции. Но их можно использовать также для порождения значений безо всякого источника данных. В следующем разделе приведен пример.

Но сначала будет уместно краткое обсуждение в чем-то пересекающихся понятий *итератора* и *генератора*.

Сравнение итераторов и генераторов

В официальной документации и кодовой базе Python терминология, относящаяся к итераторам и генераторам, противоречива и постоянно изменяется. Я принял для себя следующие определения:

итератор

> Общий термин, обозначающий любой объект, который реализует метод `__next__`. Итераторы предназначены для порождения данных, потребляемых клиентским кодом, т. е. кодом, который управляет итератором посредством цикла `for` или другой итеративной конструкции либо путем явного вызова функции `next(it)` для итератора – хотя такое явное использование встречается гораздо реже. На практике большинство итераторов, встречающихся в Python, являются генераторами.

генератор

> Итератор, построенный компилятором Python. Для создания генератора мы не реализуем метод `__next__`. Вместо этого используется ключевое слово `yield`, в результате чего получается *генераторная функция*, т. е. фабрика *объектов-генераторов*. *Генераторное выражение* – еще один способ построить объект-генератор. Объекты-генераторы предоставляют метод `__next__`, т. е. являются генераторами. Начиная с версии Python 3.5 в язык включены также асинхронные генераторы, объявляемые с помощью конструкции `async def`. Мы будем изучать их в главе 21.

В глоссарии Python недавно появился термин генераторный итератор (*https://docs.python.org/3/glossary.html#term-generator-iterator*), так называют объекты, построенные генераторными функциями, тогда как в статье о генераторных выражениях (*https://docs.python.org/3/glossary.html#term-generator-expression*) говорится, что они возвращают «итератор». Но в обоих случаях, если верить Python, возвращаются объекты-генераторы:

```
>>> def g():
...     yield 0
...
>>> g()
<generator object g at 0x10e6fb290>
>>> ge = (c for c in 'XYZ')
>>> ge
<generator object <genexpr> at 0x10e936ce0>
>>> type(g()), type(ge)
(<class 'generator'>, <class 'generator'>)
```

Генератор арифметической прогрессии

Классический паттерн Итератор относится к обходу некоторой структуры данных. Но стандартный интерфейс, основанный на методе извлечения следующего элемента ряда, полезен и тогда, когда элементы порождаются «на лету», а не выбираются из коллекции. Например, встроенная функция `range` генерирует ограниченную арифметическую прогрессию целых чисел. А что, если нужно сгенерировать арифметическую прогрессию чисел произвольного типа, а не только целых?

В примере 17.11 показано несколько тестов класса `ArithmeticProgression`, который мы вскоре напишем. Конструктор имеет сигнатуру `ArithmeticProgression(begin, step[, end])`. Полная сигнатура функции `range()` имеет вид `range(start, stop[, step])`. Я выбрал другую сигнатуру, потому что для арифметической прогрессии шаг `step` обязателен, а конечное значение `end` – нет. Кроме того, я заменил имена аргументов `start/stop` на `begin/end`, дав понять, что сигнатура поменялась. Во всех тестах в примере 17.11 я вызываю конструктор `list()` для просмотра сгенерированных значений.

Пример 17.11. Демонстрация класса `ArithmeticProgression`

```
>>> ap = ArithmeticProgression(0, 1, 3)
>>> list(ap)
[0, 1, 2]
>>> ap = ArithmeticProgression(1, .5, 3)
>>> list(ap)
[1.0, 1.5, 2.0, 2.5]
>>> ap = ArithmeticProgression(0, 1/3, 1)
>>> list(ap)
[0.0, 0.3333333333333333, 0.6666666666666666]
>>> from fractions import Fraction
>>> ap = ArithmeticProgression(0, Fraction(1, 3), 1)
>>> list(ap)
[Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
>>> from decimal import Decimal
>>> ap = ArithmeticProgression(0, Decimal('.1'), .3)
>>> list(ap)
[Decimal('0'), Decimal('0.1'), Decimal('0.2')]
```

Отметим, что числа в получающейся арифметической прогрессии имеют тот же тип, что `begin + step`, – согласно общим правилам приведения числовых типов в Python. В примере 17.11 мы видим список чисел типа `int`, `float`, `Fraction` и `Decimal`. В примере 17.12 показана реализация класса `ArithmeticProgression`.

Пример 17.12. Класс `ArithmeticProgression`

```
class ArithmeticProgression:

    def __init__(self, begin, step, end=None): ❶
        self.begin = begin
        self.step = step
        self.end = end # None -> "бесконечный" ряд

    def __iter__(self):
        result_type = type(self.begin + self.step) ❷
```

```
        result = result_type(self.begin) ❸
        forever = self.end is None ❹
        index = 0
        while forever or result < self.end: ❺
            yield result ❻
            index += 1
            result = self.begin + self.step * index ❼
```

❶ `__init__` требует двух аргументов: `begin` и `step`. Аргумент `end` необязательный, если он равен `None`, ряд будет неограниченным.
❷ Получить тип суммы `self.begin` и `self.step`. Например, если одно имеет тип `int`, а другое `float`, то типом результата будет `float`.
❸ Эта строка порождает значение `result`, равное `self.begin`, но приведенное к типу последующих слагаемых[1].
❹ Для большей понятности я завел флаг `forever`, который равен `True`, если атрибут `self.end` равен `None`, в этом случае получается неограниченный ряд.
❺ Этот цикл продолжается вечно или пока значение `result` не окажется больше или равно `self.end`. По выходе из цикла завершается и функция.
❻ Порождается текущее значение `result`.
❼ Вычисляется следующий потенциальный результат. Возможно, он никогда не будет отдан, потому что цикл `while` завершится раньше.

В последней строке я, вместо того чтобы прибавлять к `result` значение `self.step` на каждой итерации, решил игнорировать предыдущее значение `result` и каждый раз вычислять `result` заново путем сложения `self.begin` с величиной `self.step`, умноженной на `index`. Это уменьшает накопление погрешности при работе с числами с плавающей точкой. Простые эксперименты ясно показывают разницу:

```
>>> 100 * 1.1
110.00000000000001
>>> sum(1.1 for _ in range(100))
109.99999999999982
>>> 1000 * 1.1
1100.0
>>> sum(1.1 for _ in range(1000))
1100.0000000000086
```

Показанный выше класс `ArithmeticProgression` работает, как и было задумано, и дает понятный пример использования генераторной функции для реализации специального метода `__iter__`. Однако если единственная цель класса – сконструировать генератор в методе `__iter__`, то класс можно свести к генераторной функции. Ведь генераторная функция – это не что иное, как фабрика генераторов.

[1] В Python 2 была встроенная функция `coerce()`, но в Python 3 ее убрали, сочтя лишней, т. к. правила приведения числовых типов неявно встроены в методы арифметических операторов. Поэтому единственный способ, который я смог придумать для приведения начального значения к тому же типу, что остальные члены ряда, – выполнить сложение и воспользоваться его типом для преобразования результата. Я задал этот вопрос в списке рассылки Python-list и получил отличный ответ от Стивена Д'Апрано (*https://marc.info/?l=python-list&m=141826925106951&w=2*).

В примере 17.13 показана генераторная функция `aritprog_gen`, которая делает то же самое, что класс `ArithmeticProgression`, но короче. Все тесты в примере 17.11 проходят, если вызывать `aritprog_gen` вместо `ArithmeticProgression`[1].

Пример 14.12. Генераторная функция `aritprog_gen`

```
def aritprog_gen(begin, step, end=None):
    result = type(begin + step)(begin)
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
        result = begin + step * index
```

Пример 17.13, конечно, элегантный, но не забывайте: в стандартной библиотеке немало готовых генераторов, и в следующем разделе мы покажем еще более короткую реализацию с использованием модуля `itertools`.

Построение арифметической прогрессии с помощью itertools

Модуль `itertools` в версии Python 3.10 содержит 20 генераторных функций, которые можно комбинировать разными интересными способами.

Например, функция `itertools.count` возвращает генератор, порождающий числа. Без аргументов порождается ряд целых чисел, начиная с 0. А если задать аргументы `start` и `step`, то получится результат, очень похожий на тот, что дают наши функции `aritprog_gen`:

```
>>> import itertools
>>> gen = itertools.count(1, .5)
>>> next(gen)
1 >>> next(gen)
1.5
>>> next(gen)
2.0
>>> next(gen)
2.5
```

`itertools.count` никогда не останавливается, поэтому, обрабатывая вызов `list(count())`, Python попытается построить список, не помещающийся в оперативную память, и ваша машина начнет сварливо брюзжать задолго до того, как вызов завершится ошибкой.

С другой стороны, существует функция `itertools.takewhile`: она порождает генератор, который потребляет другой генератор и останавливается, когда заданный предикат станет равен `False`. Объединив обе функции вместе, мы можем написать:

[1] Каталог *14-it-generator/ directory* в репозитории кода к этой книге (*https://github.com/fluentpython/example-code-2e*) содержит тесты, а также скрипт *aritprog_runner.py*, который прогоняет все тесты для различных вариантов скриптов *aritprog*.py*.

```
>>> gen = itertools.takewhile(lambda n: n < 3, itertools.count(1, .5))
>>> list(gen)
[1, 1.5, 2.0, 2.5]
```

Благодаря использованию `takewhile` и `count` мы получаем еще более короткую реализацию, показанную в примере 17.14.

Пример 17.14. aritprog_v3.py: работает, как предыдущие варианты функции `aritprog_gen`

```
import itertools

def aritprog_gen(begin, step, end=None):
    first = type(begin + step)(begin)
    ap_gen = itertools.count(first, step)
    if end is None:
        return ap_gen
    return itertools.takewhile(lambda n: n < end, ap_gen)
```

Отметим, что функция `aritprog_gen` в примере 17.14 не является генераторной функцией: в ней нет слова `yield`. Но она возвращает генератор, как и генераторная функция.

Напомним, однако, что `itertools.count` прибавляет на каждом шаге `step`, поэтому порождаемый ей ряд чисел с плавающей точкой не такой точный, как в примере 17.13.

Посыл, содержащийся в примере 17.14, прост: реализуя генераторы, нужно знать, что уже есть в стандартной библиотеке, иначе велики шансы изобрести велосипед. Вот почему в следующем разделе мы рассмотрим несколько готовых генераторных функций.

Генераторные функции в стандартной библиотеке

В стандартной библиотеке есть много генераторов: от объектов построчного чтения текстового файла до восхитительной функции `os.walk` (*https://docs.python.org/3/library/os.html#os.walk*), которая обходит дерево каталогов и отдает имена файлов, в результате чего рекурсивный поиск оказывается не сложнее обычного цикла `for`.

Генераторная функция `os.walk` впечатляет, но в этом разделе я сконцентрируюсь на функциях общего назначения, которые принимают произвольные итерируемые объекты в качестве аргументов и возвращают генераторы, порождающие выборку, результаты вычислений или элементы в другом порядке. В следующих таблицах я перечислил два десятка таких функций, встроенных и находящихся в модулях `itertools` и `functools`. Для удобства они сгруппированы по общей функциональности вне зависимости от того, где находятся.

Первая группа – фильтрующие генераторные функции: они отдают подмножество элементов, порождаемых входным итерируемым объектом, не изменяя сами элементы. Как и функция `takewhile`, большинство перечисленных в табл. 17.1 функций принимают предикат – булеву функцию с одним аргументом, которая применяется к каждому входному элементу и определяет, нужно ли отдавать его на выходе.

Таблица 17.1. Фильтрующие генераторные функции

Модуль	Функция	Описание
itertools	compress(it, selector_it)	Потребляет параллельно два итерируемых объекта; отдает элемент it, когда соответствующий элемент selector_it принимает похожее на истину значение
itertools	dropwhile(predicate, it)	Потребляет it, пропуская элементы, пока predicate принимает похожее на истину значение, а затем отдает все оставшиеся элементы (больше никаких проверок не делается)
Встроенная	filter(predicate, it)	Применяет предикат к каждому элементу итерируемого объекта, отдавая элемент, если predicate(item) принимает похожее на истину значение; если predicate равен None, отдаются только элементы, принимающие похожее на истину значение
itertools	filterfalse(predicate, it)	То же, что filter, но логика инвертирована: отдаются элементы, для которых предикат принимает похожее на ложь значение
itertools	islice(it, stop) или islice(it, start, stop, step=1)	Отдает элементы из среза it по аналогии с s[:stop] или s[start:stop:step], только it может быть произвольным итерируемым объектом, а операция ленивая
itertools	takewhile(predicate, it)	Отдает элементы, пока predicate принимает похожее на истину значение, затем останавливается, больше никаких проверок не делается

В распечатке сеанса оболочки ниже приведены примеры применения всех функций из табл. 17.1.

Пример 17.15. Примеры фильтрующих генераторных функций

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
...
>>> list(filter(vowel, 'Aardvark'))
['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1, 0, 1, 1, 0, 1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']
```

Следующая группа – отображающие генераторы: они отдают элементы, вычисленные для каждого элемента входного итерируемого объекта – или нескольких таких объектов, как в случае `map` и `starmap`[1]. Генераторы, перечисленные в табл. 17.2, отдают по одному результату для каждого элемента входного итерируемого объекта. Если на вход подается несколько итерируемых объектов, то процесс прекращается, как только будет исчерпан хотя бы один из них.

Таблица 17.2. Отображающие генераторные функции

Модуль	Функция	Описание
`itertools`	`accumulate(it, [func])`	Отдает накопленные суммы; если задана функция `func`, то отдает результат применения ее к первой паре элементов, затем к первому результату и следующему элементу и т. д.
Встроенная	`enumerate(iterable, start=0)`	Отдает 2-кортежи вида `(index, item)`, где `index` начинается со значения `start`, а `item` извлекается из `iterable`
Встроенная	`map(func, it1, [it2, …, itN])`	Применяет `func` к каждому элементу `it` и отдает результат; если задано `N` итерируемых объектов, то `func` должна принимать `N` аргументов, и все итерируемые объекты обходятся параллельно
`itertools`	`starmap(func, it)`	Применяет `func` к каждому элементу `it` и отдает результат; входной итерируемый объект должен отдавать итерируемые элементы `iit`, а `func` вызывается в виде `func(*iit)`

В примере 17.16 демонстрируется несколько применений функции `itertools.accumulate`.

Пример 17.16. Примеры применения генераторной функции `itertools.accumulate`

```
>>> import itertools
>>> list(itertools.accumulate(sample)) ❶
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(itertools.accumulate(sample, min)) ❷
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max)) ❸
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul)) ❹
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800] ❺
```

❶ Частичные суммы.
❷ Частичные минимумы.
❸ Частичные максимумы.
❹ Частичные произведения.
❺ Факториалы от 1! до 10!.

[1] Здесь термин «отображение» никак не связан со словарями, а имеет отношение к встроенной функции `map`.

Применение остальных функций из табл. 17.2 иллюстрируется в примере 17.17.

Пример 17.17. Примеры применения отображающих генераторных функций

```
>>> list(enumerate('albatroz', 1)) ❶
[(1, 'a'), (2, 'l'), (3, 'b'), (4, 'a'), (5, 't'), (6, 'r'), (7,'o'), (8, 'z')]
>>> import operator
>>> list(map(operator.mul, range(11), range(11))) ❷
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8])) ❸
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8])) ❹
[(0, 2), (1, 4), (2, 8)]
>>> import itertools
>>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1))) ❺
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'ooooooo',
'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b / a,
... enumerate(itertools.accumulate(sample), 1))) ❻
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333,
5.0, 4.375, 4.888888888888889, 4.5]
```

❶ Количество букв в слове, начальное значение 1.
❷ Квадраты целых чисел от 0 до 10.
❸ Перемножение целых чисел из двух параллельных итерируемых объектов; операция заканчивается, когда будет достигнут конец более короткого объекта.
❹ То же самое делает встроенная функция `zip`.
❺ Повторение каждой буквы слова столько раз, каков номер ее позиции. Первая буква повторяется 1 раз.
❻ Частичные средние.

Далее идет группа объединяющих генераторов – все они отдают элементы из нескольких входных итерируемых объектов. Функции `chain` и `chain.from_iterable` обходят входные итерируемые объекты последовательно (один за другим), а `product`, `zip` и `zip_longest` – параллельно.

Таблица 17.3. Генераторные функции, объединяющие несколько входных итерируемых объектов

Модуль	Функция	Описание
`itertools`	`chain(it1, …, itN)`	Отдает все элементы из `it1`, затем из `it2` и т. д.
`itertools`	`chain.from_iterable(it)`	Отдает все элементы из каждого итерируемого объекта, порождаемого `it`, перебирая их один за другим; `it` должен порождать итерируемые объекты, например это может быть список итерируемых объектов
`itertools`	`product(it1, …, itN, repeat=1)`	Декартово произведение: отдает *N*-кортежи, полученные путем комбинирования элементов из каждого входного итерируемого объекта, – так, как это делалось бы с помощью вложенных циклов `for`; аргумент `repeat` позволяет обходить входные итерируемые объекты более одного раза

Окончание табл. 17.3

Модуль	Функция	Описание
Встроенная	`zip(it1, …, itN, strict=False)`	Отдает *N*-кортежи, построенные из элементов, которые берутся параллельно из входных итерируемых объектов; операция прекращается по исчерпании самого короткого объекта, если только не задан параметр `strict=True`[a]
`itertools`	`zip_longest(it1, …, itN, fillvalue=None)`	Отдает *N*-кортежи, построенные из элементов, которые берутся параллельно из входных итерируемых объектов; операция прекращается по исчерпании самого длинного объекта, а вместо недостающих элементов подставляется значение `fillvalue`

[a] Чисто именованный параметр `strict` появился в версии Python 3.10. Если `strict=True`, то в случае, когда итерируемые объекты имеют разную длину, возбуждается исключение `ValueError`. По умолчанию значение равно `False` ради обратной совместимости.

В примере 17.18 показано использование генераторных функций `itertools.chain`, `zip` и родственных им. Напомним, что название функции `zip` происходит от слова zipper (застежка-молния) и не имеет никакого отношения к алгоритму сжатия. Обе функции, `zip` и `itertools.zip_longest`, были впервые продемонстрированы во врезке «Удивительная функция zip» в главе 12.

Пример 17.18. Примеры применения объединяющих генераторных функций

```
>>> list(itertools.chain('ABC', range(2))) ❶
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC'))) ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC'))) ❸
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5), [10, 20, 30, 40])) ❹
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5))) ❺
[('A', 0), ('B', 1), ('C', 2), (None, 3), (None, 4)]
>>> list(itertools.zip_longest('ABC', range(5), fillvalue='?')) ❻
[('A', 0), ('B', 1), ('C', 2), ('?', 3), ('?', 4)]
```

❶ `chain` обычно вызывается с двумя и более итерируемыми объектами.

❷ При вызове с одним итерируемым объектом `chain` не делает ничего полезного.

❸ Но `chain.from_iterable` берет каждый элемент из итерируемого объекта и сцепляет их в последовательность, при условии что каждый элемент сам является итерируемым объектом.

❹ `zip` может параллельно обходить произвольное количество итерируемых объектов, но генератор останавливается, как только один из них будет исчерпан. В Python ⩾ 3.10, если задан аргумент `strict=True` и один итератор исчерпывается раньше остальных, возбуждается исключение `ValueError`.

❺ `itertools.zip_longest` работает, как `zip`, но не останавливается, пока не будут исчерпаны все итерируемые объекты; вместо недостающих элементов в данном случае подставляется `None`.

❻ Аргумент `fillvalue` задает подстановочное значение.

Функция `itertools.product` дает ленивый способ вычисления декартовых произведений, в разделе «Декартовы произведения» главы 2 мы строили их с помощью списковых включений с несколькими фразами `for`. Для ленивого порождения декартовых произведений также можно использовать генераторные выражения с несколькими фразами `for`. В примере 17.19 демонстрируется функция `itertools.product`.

Пример 17.19. Примеры применения генераторной функции `itertools.product`

```
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)] ❶
>>> suits = 'spades hearts diamonds clubs'.split()
>>> list(itertools.product('AK', suits)) ❷
[('A', 'spades'), ('A', 'hearts'), ('A', 'diamonds'), ('A',
'clubs'),
('K', 'spades'), ('K', 'hearts'), ('K', 'diamonds'), ('K',
'clubs')]
>>> list(itertools.product('ABC')) ❸
[('A',), ('B',), ('C',)]
>>> list(itertools.product('ABC', repeat=2)) ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'),
('B', 'C'), ('C', 'A'), ('C', 'B'), ('C', 'C')]
>>> list(itertools.product(range(2), repeat=3))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
(1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> rows = itertools.product('AB', range(2), repeat=2)
>>> for row in rows: print(row)
...
('A', 0, 'A', 0)
('A', 0, 'A', 1)
('A', 0, 'B', 0)
('A', 0, 'B', 1)
('A', 1, 'A', 0)
('A', 1, 'A', 1)
('A', 1, 'B', 0)
('A', 1, 'B', 1)
('B', 0, 'A', 0)
('B', 0, 'A', 1)
('B', 0, 'B', 0)
('B', 0, 'B', 1)
('B', 1, 'A', 0)
('B', 1, 'A', 1)
('B', 1, 'B', 0)
('B', 1, 'B', 1)
```

❶ Декартово произведение строки `str` из трех символов и диапазона `range` из двух целых чисел дает шесть кортежей (потому что `3 * 2 = 6`).

❷ Произведение двух достоинств карт (`'AK'`) и четырех мастей дает ряд из восьми кортежей.

❸ Если задан один итерируемый объект, то `product` порождает ряд 1-кортежей, что не очень полезно.

❹ Но если дополнительно задан именованный аргумент `repeat=N`, то `product` обходит каждый входной итерируемый объект `N` раз.

Некоторые генераторные функции расширяют свой аргумент, отдавая более одного значения для каждого входного элемента. Они перечислены в табл. 17.4.

Таблица 17.4. Генераторные функции, расширяющие каждый входной элемент в несколько выходных

Модуль	Функция	Описание
itertools	combinations(it, out_len)	Отдает комбинации out_len элементов из элементов, отдаваемых it
itertools	combinations_with_replacement (it, out_len)	Отдает комбинации out_len элементов из элементов, отдаваемых it, включая комбинации с повторяющимися элементами
itertools	count(start=0, step=1)	Отдает числа, начиная с start с шагом step
itertools	cycle(it)	Отдает элементы из it, запоминая копию каждого, после чего отдает всю последовательность еще раз – и так до бесконечности
itertools	pairwise(it)	Отдает пары последовательных элементов из it с перекрытием[a]
itertools	permutations(it, out_len=None)	Отдает перестановки out_len элементов из элементов, отдаваемых it; по умолчанию out_len равно len(list(it))
itertools	repeat(item, [times])	Повторно отдает заданный элемент – times раз или бесконечно, если этот аргумент не задан

[a] itertools.pairwise добавлена в версии Python 3.10.

Функции count и repeat из модуля itertools возвращают генераторы, которые извлекают элементы «из воздуха»: ни одна из них не принимает итерируемый объект в качестве аргумента. Как работает функция itertools.count, мы видели в разделе «Построение арифметической прогрессии с помощью itertools» выше. Генератор cycle создает внутреннюю копию входного итерируемого объекта и в бесконечном цикле отдает его элементы снова и снова. В примере 17.20 показаны примеры применения count, cycle, pairwise и repeat.

Пример 17.20. Функции count, cycle, pairwise и repeat

```
>>> ct = itertools.count() ❶
>>> next(ct) ❷
0
>>> next(ct), next(ct), next(ct) ❸
(1, 2, 3)
>>> list(itertools.islice(itertools.count(1, .3), 3)) ❹
[1, 1.3, 1.6]
>>> cy = itertools.cycle('ABC') ❺
>>> next(cy)
'A'
>>> list(itertools.islice(cy, 7)) ❻
['B', 'C', 'A', 'B', 'C', 'A', 'B']
>>> list(itertools.pairwise(range(7)))
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)] ❼
>>> rp = itertools.repeat(7) ❽
>>> next(rp), next(rp)
(7, 7)
```

```
>>> list(itertools.repeat(8, 4)) ❾
[8, 8, 8, 8]
>>> list(map(operator.mul, range(11), itertools.repeat(5))) ❿
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

❶ `count` создает генератор `ct`.
❷ Получить от `ct` первый элемент.
❸ Построить из `ct` список невозможно, т. к. `ct` никогда не останавливается, поэтому я просто получаю следующие три элемента.
❹ Построить список с помощью генератора `count` можно, если он ограничен с помощью функции `islice` или `takewhile`.
❺ Построить генератор `cycle` из `'ABC'` и получить от него первый элемент – `'A'`.
❻ Список можно построить, только если наложить ограничение с помощью `islice`; здесь извлекаются следующие семь элементов.
❼ Для каждого элемента входного итерируемого объекта `pairwise` отдает 2-кортеж, содержащий этот и следующий за ним элементы, если следующий имеется. Доступна начиная с Python 3.10.
❽ Построить генератор `repeat`, который вечно отдает число 7.
❾ Генератор `repeat` можно ограничить, передав аргумент `times`: данном случае число 8 отдается 4 раза.
❿ Типичное применение `repeat`: подстановка фиксированного аргумента в функцию `map`: в данном случае подставляется множитель 5.

Генераторные функции `combinations`, `combinations_with_replacement` и `permutations` – вместе с `product` – в документации `itertools` называются *комбинаторными генераторами* (*https://docs.python.org/3/library/itertools.html*). Существует тесная связь между `itertools.product` и остальными комбинаторными функциями (см. пример 17.21).

Пример 17.21. Комбинаторные генераторные функции отдают несколько значений для каждого входного элемента

```
>>> list(itertools.combinations('ABC', 2)) ❶
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.combinations_with_replacement('ABC', 2)) ❷
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
>>> list(itertools.permutations('ABC', 2)) ❸
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>> list(itertools.product('ABC', repeat=2)) ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'),
('C', 'A'), ('C', 'B'), ('C', 'C')]
```

❶ Все комбинации длины `len()==2` из элементов строки `'ABC'`; порядок элементов в сгенерированных кортежах неважен (они могли бы быть и множествами).
❷ Все комбинации длины `len()==2` из элементов строки `'ABC'`, включая комбинации с повторяющимися элементами.
❸ Все перестановки длины `len()==2` из элементов строки `'ABC'`; порядок элементов в сгенерированных кортежах важен.
❹ Декартово произведение `'ABC'` и `'ABC'` (это результат задания параметра `repeat=2`).

Последняя рассматриваемая в этом разделе группа генераторных функций предназначена для того, чтобы отдавать все элементы входных итерируемых объектов, но в каком-то другом порядке. Следующие две функции возвращают несколько генераторов: `itertools.groupby` и `itertools.tee`. Другая генераторная функция из этой группы, встроенная функция `reversed`, – единственная из описанных в этом разделе, которая принимает не произвольный итерируемый объект, а только последовательности. Это и понятно, ведь `reversed` отдает элементы в обратном порядке, а это возможно только для последовательности известной длины. Но накладных расходов на создание инвертированной копии последовательности эта функция не несет – она возвращает элементы по запросу. Я поместил функцию `itertools.product` в одну группу с объединяющими генераторами в табл. 17.3, потому что все они обходят более одного итерируемого объекта, тогда как генераторы, перечисленные в табл. 17.5, принимают не больше одного такого объекта.

Таблица 17.5. Реорганизующие генераторные функции

Модуль	Функция	Описание
`itertools`	`groupby(it, key=None)`	Порождает 2-кортежи вида (`key`, `group`), где `key` – критерий группировки, а `group` – генератор, отдающий элементы группы
Встроенная	`reversed(seq)`	Отдает элементы `seq` в обратном порядке, от последнего к первому; аргумент `seq` должен быть последовательностью или реализовывать специальный метод `__reversed__`
`itertools`	`tee(it, n=2)`	Отдает кортеж *n* генераторов, каждый из которых независимо отдает элементы входного итерируемого объекта

В примере 17.22 демонстрируется использование функций `itertools.groupby` и `reversed`. Отметим, что `itertools.groupby` ожидает, что входной итерируемый объект отсортирован в соответствии с критерием группировки или, по крайней мере, что элементы, удовлетворяющие этому критерию, идут подряд, пусть даже и не по порядку. Технический рецензент Мирослав Седивы предложил такой пример использования: можно отсортировать объекты типа `datetime` хронологически, а затем сгруппировать по дню недели, тогда получится группа данных за понедельник, потом за вторник и т. д., а затем снова за понедельник (следующей недели) и т. д.

Пример 17.22. `itertools.groupby`

```
>>> list(itertools.groupby('LLLLAAGGG')) ❶
[('L', <itertools._grouper object at 0x102227cc0>),
('A', <itertools._grouper object at 0x102227b38>),
('G', <itertools._grouper object at 0x102227b70>)]
>>> for char, group in itertools.groupby('LLLLAAAGG'): ❷
... print(char, '->', list(group))
...
L -> ['L', 'L', 'L', 'L']
A -> ['A', 'A',]
G -> ['G', 'G', 'G']
>>> animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
```

```
...             'bat', 'dolphin', 'shark', 'lion']
>>> animals.sort(key=len) ❸
>>> animals
['rat', 'bat', 'duck', 'bear', 'lion', 'eagle', 'shark',
'giraffe', 'dolphin']
>>> for length, group in itertools.groupby(animals, len): ❹
...     print(length, '->', list(group))
...
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
>>> for length, group in itertools.groupby(reversed(animals), len): ❺
...     print(length, '->', list(group))
...
7 -> ['dolphin', 'giraffe']
5 -> ['shark', 'eagle']
4 -> ['lion', 'bear', 'duck']
3 -> ['bat', 'rat']
>>>
```

❶ `groupby` отдает кортежи `(key, group_generator)`.

❷ Для работы с генераторами, порожденными `groupby`, необходимы вложенные итерации: в данном случае внешний цикл `for` и внутренний конструктор `list`.

❸ Для использования `groupby` входной объект должен быть отсортирован; в данном случае слова отсортированы по длине.

❹ Еще один цикл по парам (`key, group`), чтобы вывести ключ и развернуть группу в список.

❺ Здесь генератор `reversed` используется для обхода `animals` справа налево.

Последняя генераторная функция в этой группе, `iterator.tee`, обладает уникальным поведением: она порождает несколько генераторов для одного входного итерируемого объекта, каждый из которых отдает все элементы этого объекта. Эти генераторы можно потреблять независимо, как показано в примере 17.23.

Пример 17.23. `itertools.tee` порождает несколько генераторов, каждый из которых отдает все элементы входного итерируемого объекта

```
>>> list(itertools.tee('ABC'))
[<itertools._tee object at 0x10222abc8>, <itertools._tee object at
0x10222ac08>]
>>> g1, g2 = itertools.tee('ABC')
>>> next(g1)
'A'
>>> next(g2)
'A'
>>> next(g2)
'B'
>>> list(g1)
['B', 'C']
>>> list(g2)
['C']
>>> list(zip(*itertools.tee('ABC')))
[('A', 'A'), ('B', 'B'), ('C', 'C')]
```

Отметим, что в нескольких примерах из этого раздела использовались комбинации генераторных функций. Это возможно, потому что все они принимают генераторы в качестве аргументов и возвращают генераторы.

Теперь рассмотрим еще одну группу функций для работы с итераторами из стандартной библиотеки.

Функции редуцирования итерируемого объекта

Все функции, перечисленные в табл. 17.6, принимают итерируемый объект и возвращают единственный результат. Их называют «редуцирующими», «сворачивающими» или «аккумулирующими». На самом деле все эти функции можно было бы реализовать с помощью `functools.reduce`, но они сделаны встроенными, чтобы было проще решать часто встречающиеся задачи. Более пространное рассмотрение `functools.reduce` было приведено в разделе «Vector, попытка № 4: хеширование и ускорение оператора ==» главы 12.

Для `all` и `any` произведена важная оптимизация, которая при использовании `reduce` была бы невозможна: эти функции закорочены (т. е. прекращают обход итератора, как только результат становится известен). См. последний тест функции `any` в примере 17.24.

Таблица 17.6. Встроенные функции, которые читают итерируемый объект и возвращают одиночное значение

Модуль	Функция	Описание
Встроенная	`all(it)`	Возвращает `True`, если все элементы `it` принимают похожее на истину значение, в противном случае `False`; `all([])` возвращает `True`
Встроенная	`any(it)`	Возвращает `True`, если хотя бы один элемент `it` принимает похожее на истину значение, в противном случае `False`; `any([])` возвращает `False`
Встроенная	`max(it, [key=,] [default=])`	Возвращает максимальный элемент `it`[a]; `key` – функция порядка, как в `sorted`; значение `default` возвращается, если итерируемый объект пуст
Встроенная	`min(it, [key=,] [default=])`	Возвращает минимальный элемент `it`[b]; `key` – функция порядка, как в `sorted`; значение `default` возвращается, если итерируемый объект пуст
`functools`	`reduce(func, it, [initial])`	Возвращает результат выполнения следующей процедуры: функция `func` применяется к первым двум элементам, затем к результату и третьему элементу и т. д. Если задан аргумент `initial`, то он образует начальную пару вместе с первым элементом
Встроенная	`sum(it, start=0)`	Сумма всех элементов `it`, к которой может быть прибавлено значение `start`, если оно задано (для получения большей точности при сложении чисел с плавающей точкой пользуйтесь функцией `math.fsum`)

[a] Может также вызываться в виде `max(arg1, arg2, …, [key=?])`, тогда возвращается максимальный аргумент.

[b] Может также вызываться в виде `min(arg1, arg2, …, [key=?])`, тогда возвращается минимальный аргумент.

Работа `all` и `any` демонстрируется в примере 17.24.

Пример 17.24. Результаты применения `all` и `any` к некоторым последовательностям

```
>>> all([1, 2, 3])
True
>>> all([1, 0, 3])
False
>>> all([])
True
>>> any([1, 2, 3])
True
>>> any([1, 0, 3])
True
>>> any([0, 0.0])
False
>>> any([])
a
b
False
>>> g = (n for n in [0, 0.0, 7, 8])
>>> any(g) ❶
True
>>> next(g) ❷
8
```

❶ `any` продолжает итерации, пока `g` не отдаст `7`; тогда `any` останавливается и возвращает `True`.
❷ Именно поэтому `8` осталось.

Встроенная функция `sorted` также принимает итерируемый объект и возвращает нечто иное. В отличие от генераторной функции `reversed`, `sorted` строит и возвращает настоящий список. В конце концов, каждый элемент входного итерируемого объекта можно прочитать, а раз так, то их можно и отсортировать, причем сортировке подвергается список `list`, а значит, его `sorted` и возвращает. Я упомянул `sorted` в этом месте, потому что она все-таки принимает произвольный итерируемый объект.

Конечно, `sorted` и редуцирующие функции работают только с конечными итерируемыми объектами. В противном случае они будут без конца получать элементы и никогда не вернут результат.

Если вы дочитали до этого места, то уже знаете о самом важном и полезном материале данной главы. В остальных разделах рассматриваются продвинутые функции генераторов, которые большинству из нас приходится видеть или использовать нечасто. Речь идет о конструкции `yield from` и классических сопрограммах.

Есть также разделы, посвященные аннотациям типов для итерируемых объектов, итераторов и классических сопрограмм.

Конструкция `yield from` дает новый способ комбинирования генераторов.

YIELD FROM И СУБГЕНЕРАТОРЫ

Выражение `yield from` было добавлено в версию Python 3.3 и позволяет генератору делегировать работу субгенератору.

Раньше мы использовали вложенные циклы `for`, когда генераторная функция должна была отдавать значения, порождаемые другим генератором.

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...
>>> def gen():
...     yield 1
...     for i in sub_gen():
...         yield i
...     yield 2
...
>>> for x in gen():
...     print(x)
...
1
1.1
1.2
2
```

Тот же результат можно получить с помощью `yield from`, как показано в примере 17.25.

Пример 17.25. Тест, демонстрирующий работу `yield from`

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...
>>> def gen():
...     yield 1
...     yield from sub_gen()
...     yield 2
...
>>> for x in gen():
...     print(x)
...
1
 1.1
1.2
2
```

В примере 17.25 цикл `for` – это *клиентский код*, `gen` – *делегирующий генератор*, а `sub_gen` – *субгенератор*. Заметим, что `yield from` приостанавливает `gen`, после чего `sub_gen` работает, пока не исчерпается. Значения, отдаваемые `sub_gen`, передаются сквозь `gen` напрямую клиентскому циклу `for`. Тем временем `gen` пребывает в нирване и не видит проходящих сквозь него значений. И лишь когда `sub_gen` закончит работу, `gen` возобновляется.

Если субгенератор содержит предложение `return`, возвращающее значение, то это значение может быть перехвачено делегирующим генератором, если он включит конструкцию `yield from` в состав выражения. См. пример 17.26.

Пример 17.26. `yield from` получает значение, возвращенное субгенератором

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...     return 'Done!'
...
>>> def gen():
...     yield 1
...     result = yield from sub_gen()
...     print('<--', result)
...     yield 2
...
>>> for x in gen():
...     print(x)
...
1
1.1
1.2
<-- Done!
2
```

Познакомившись с основами `yield from`, рассмотрим два простых, но практически полезных примера ее использования.

Изобретаем chain заново

В табл. 17.3 мы видели, что в модуле `itertools` есть генератор `chain`, который отдает элементы нескольких итерируемых объектов, сначала перебирая первый, потом второй и так далее до последнего. Вот как выглядит доморощенная реализация такого сцепляющего генератора `chain` с помощью цикла `for` на Python[1]:

```
>>> def chain(*iterables):
...     for it in iterables:
...         for i in it:
...         yield i
...
>>> s = 'ABC'
>>> r = range(3)
>>> list(chain(s, r))
['A', 'B', 'C', 0, 1, 2]
```

Этот генератор `chain` дает шанс поработать каждому полученному итерируемому объекту по очереди. Внутренний цикл можно заменить выражением `yield from`, как показано в сеансе консоли ниже:

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i
...
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

Конструкция `yield from` здесь используется правильно, и код действительно смотрится лучше, но в таком виде это всего лишь синтаксический сахар, который мало что дает. Давайте напишем более интересный пример.

[1] Функция `chain`, как и большая часть функций из модуля `itertools`, написана на C.

Обход дерева

В этом разделе мы посмотрим, как можно использовать `yield from` в скрипте для обхода древовидной структуры. Будем продвигаться малыми шажками.

Древовидной структурой в этом примере будет иерархия исключений (*https://docs.python.org/3/library/exceptions.html#exception-hierarchy*) в Python. Но сам паттерн можно адаптировать для показа иерархии каталогов или любой другой древовидной структуры.

В Python 3.10 иерархия исключений начинается с `BaseException` на нулевом уровне и имеет пять уровней. Первый наш шаг – показать нулевой уровень.

Получив корневой класс, генератор `tree` в примере 17.27 отдает его имя и останавливается.

Пример 17.27. tree/step0/tree.py: отдать имя корневого класса и остановиться

```
def tree(cls):
    yield cls.__name__

def display(cls):
    for cls_name in tree(cls):
        print(cls_name)

if __name__ == '__main__':
    display(BaseException)
```

На выходе печатается единственная строка:

```
BaseException
```

Следующий шажок переводит нас на уровень 1. Генератор `tree` отдает имя корневого класса и имена всех его прямых подклассов. Имена подклассов печатаются с отступом, чтобы была видна иерархия. Вот какое представление мы хотим получить:

```
$ python3 tree.py
BaseException
    Exception
    GeneratorExit
    SystemExit
    KeyboardInterrupt
```

Это делает код в примере 17.28.

Пример 17.28. tree/step1/tree.py: отдать имя корневого класса и его прямых подклассов

```
def tree(cls):
    yield cls.__name__, 0 ❶
    for sub_cls in cls.__subclasses__(): ❷
        yield sub_cls.__name__, 1 ❸

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level ❹
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

❶ Для поддержки вывода с отступами отдавать имя класса и его уровень в иерархии.
❷ Использовать специальный метод `__subclasses__` для получения списка подклассов.
❸ Отдать имя каждого подкласса и уровень 1.
❹ Построить красную строку шириной `4 * level` пробелов. На нулевом уровне строка будет пустой.

В примере 17.29 я переработал генератор `tree`, чтобы отделить специальный случай корневого класса от подклассов, которые теперь обрабатываются в генераторе `sub_tree`. Дойдя до предложения `yield from`, генератор `tree` приостанавливается, а обязанность отдавать значения переходит к `sub_tree`.

Пример 17.29. tree/step2/tree.py: `tree` отдает имя корневого класса, а затем делегирует работу `sub_tree`

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls) ❶

def sub_tree(cls):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1 ❷

def display(cls):
    for cls_name, level in tree(cls): ❸
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException
```

❶ Делегировать `sub_tree`, чтобы тот отдал имена подклассов.
❷ Отдать имя каждого подкласса и уровень 1. Благодаря наличию `yield from sub_tree(cls)` внутри `tree` эти значения проходят мимо генераторной функции...
❸ ... и доставляются прямо сюда.

Придерживаясь методики малых шагов, я напишу простейший код, который позволит достичь уровня 2. Чтобы обойти дерево в глубину (*https://en.wikipedia.org/wiki/Depth-first_search*), я хочу после отдачи каждого узла уровня 1 отдавать непосредственные потоки этого узла уровня 2, а затем вернуться на уровень 1. Для этого можно организовать вложенный цикл `for`, как в примере 17.30.

Пример 17.30. tree/step3/tree.py: функция `sub_tree` обходит уровни 1 и 2 в глубину

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls)

def sub_tree(cls):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1
        for sub_sub_cls in sub_cls.__subclasses__():
            yield sub_sub_cls.__name__, 2
```

```
def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

Вот как выглядит результат выполнения скрипта `step3/tree.py` в примере 17.30:

```
$ python3 tree.py
BaseException
    Exception
        TypeError
        StopAsyncIteration
        StopIteration
        ImportError
        OSError
        EOFError
        RuntimeError
        NameError
        AttributeError
        SyntaxError
        LookupError
        ValueError
        AssertionError
        ArithmeticError
        SystemError
        ReferenceError
        MemoryError
        BufferError
        Warning
    GeneratorExit
    SystemExit
    KeyboardInterrupt
```

Вы уже, наверное, поняли, к чему все идет, но еще раз прибегну к методике малых шагов: мы дойдем до уровня 3, добавив еще один вложенный цикл `for`. Больше в программе ничего не изменилось, поэтому в примере 17.31 показан только генератор `sub_tree`.

Пример 17.31. Генератор `sub_tree` из скрипта `tree/step4/tree.py`

```
def sub_tree(cls):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1
        for sub_sub_cls in sub_cls.__subclasses__():
            yield sub_sub_cls.__name__, 2
            for sub_sub_sub_cls in sub_sub_cls.__subclasses__():
                yield sub_sub_sub_cls.__name__, 3
```

В примере 17.31 четко видна закономерность. В цикле `for` мы получаем подклассы уровня *N*. На каждой итерации цикла мы отдаем подкласс уровня *N*, а затем входим в следующий цикл `for`, чтобы посетить уровень $N + 1$.

В разделе «Изобретаем chain заново» мы видели, как заменить вложенный цикл `for`, управляющий генератором, конструкцией `yield from` для того же гене-

ратора. Эту идею можно применить и здесь, если переделать `sub_tree`, так чтобы она принимала параметр `level` и рекурсивно применяла к себе `yield from`, передавая текущий подкласс в качестве нового корневого класса со следующим номером уровня. См. пример 17.32.

Пример 17.32. tree/step5/tree.py: рекурсивная функция `sub_tree` может продвинуться настолько глубоко, насколько хватит памяти

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls, 1)

def sub_tree(cls, level):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, level
        yield from sub_tree(sub_cls, level+1)

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

Скрипт из примера 17.32 может обходить деревья любой глубины, единственным ограничением является предел на уровень рекурсии в Python. По умолчанию допускается 1000 рекурсивных вызовов.

В любом нормальном пособии по рекурсии подчеркивается важность базы, позволяющей избежать бесконечной рекурсии. Базой является условное ветвление, в одной из ветвей которого производится возврат без рекурсивного вызова. Базовый случай часто реализуют с помощью предложения `if`. В примере 17.32 в функции `sub_tree` нет `if`, но неявное условие есть в цикле `for`: если `cls.__subclasses__()` возвращает пустой список, то тело цикла не выполняется, так что рекурсивного вызова не будет. Базовым является случай, когда класс `cls` не имеет подклассов. Тогда `sub_tree` ничего не отдает, а просто возвращает управление.

Пример 17.32 делает то, что надо, но его можно сократить, вспомнив паттерн, который мы видели, когда дошли до уровня 3 (пример 17.31): мы отдаем подкласс уровня *N*, а затем входим во вложенный цикл `for`, чтобы посетить уровень *N* + 1. В примере 17.32 мы заменили вложенный цикл конструкцией `yield from`. Теперь можно объединить `tree` и `sub_tree` в один генератор. Пример 17.33 – последний шаг.

Пример 17.33. tree/step6/tree.py: рекурсивным вызовам `tree` передается увеличенный на 1 аргумент `level`

```
def tree(cls, level=0):
    yield cls.__name__, level
    for sub_cls in cls.__subclasses__():
        yield from tree(sub_cls, level+1)

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
```

```
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

В начале раздела «yield from и субгенераторы» мы видели, как `yield from` связывает субгенератор напрямую с клиентским кодом в обход делегирующего генератора. Эта связь становится особенно важной, когда генераторы используются в качестве сопрограмм и не только порождают, но и потребляют значения из клиентского кода, как мы увидим в разделе «Классические сопрограммы».

После этого первого знакомства с `yield from` обратимся к аннотациям типов для итерируемых объектов и итераторов.

Обобщенные итерируемые типы

В стандартной библиотеке Python много функций, принимающих итерируемые объекты в качестве аргументов. В своем коде вы можете аннотировать их, как функцию `zip_replace` из примера 8.15 с использованием типа `collections.abc.Iterable` (или `typing.Iterable`, если нужно обязательно поддержать версию Python 3.8 или более ранние; см. объяснение во врезке «Поддержка унаследованных типов и нерекомендуемые типы коллекций» в главе 8). См. пример 17.34.

Пример 17.34. replacer.py возвращает итератор по кортежам строк

```
from collections.abc import Iterable

FromTo = tuple[str, str] ❶

def zip_replace(text: str, changes: Iterable[FromTo]) -> str: ❷
    for from_, to in changes:
        text = text.replace(from_, to)
    return text
```

❶ Определить псевдоним типа; необязательно, но делает следующую аннотацию типа более понятной. Начиная с Python 3.10 `FromTo` должна иметь аннотацию типа `typing.TypeAlias`, чтобы прояснить назначение этой строки: `FromTo: TypeAlias = tuple[str, str]`.

❷ Аннотировать аргумент `changes`, показав, что он принимает итерируемый объект `Iterable`, составленный из кортежей типа `FromTo`.

Типы `Iterator` встречаются не так часто, как `Iterable`, но употреблять их в аннотациях тоже просто. В примере 17.35 показан уже знакомый нам генератор чисел Фибоначчи с аннотациями.

Пример 17.35. fibo_gen.py: `fibonacci` возвращает генератор целых чисел

```
from collections.abc import Iterator

def fibonacci() -> Iterator[int]:
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

Заметим, что тип `Iterator` используется для генераторов, оформленных в виде функций с `yield`, а также итераторов, написанных «вручную» как классы с методом `__next__`. Существует также тип `collections.abc.Generator` (и соответствующий объявленный нерекомендуемым тип `typing.Generator`), который можно использовать для аннотирования объектов-генераторов, но он слишком многословен для генераторов в роли итераторов.

Если проверить код в примере 17.36 с помощью Муру, то окажется, что тип `Iterator` в действительности является упрощенным частным случаем типа `Generator`.

Пример 17.36. itergentype.py: два способа аннотирования итераторов

```
from collections.abc import Iterator
from keyword import kwlist
from typing import TYPE_CHECKING

short_kw = (k for k in kwlist if len(k) < 5) ❶

if TYPE_CHECKING:
    reveal_type(short_kw) ❷

long_kw: Iterator[str] = (k for k in kwlist if len(k) >= 4) ❸

if TYPE_CHECKING: ❹
    reveal_type(long_kw)
```

❶ Генераторное выражение, отдающее ключевые слова Python длиной менее 5 знаков.
❷ Муру выводит: `typing.Generator[builtins.str*, None, None]`[1].
❸ Это выражение также отдает строки, но я добавил явную аннотацию типа.
❹ Выведенный тип: `typing.Iterator[builtins.str]`.

`abc.Iterator[str]` *совместим с* `abc.Generator[str, None, None]`, поэтому в примере 17.36 Муру не выдает никаких ошибок при проверке типов.

`Iterator[T]` – краткое обозначение `Generator[T, None, None]`. Обе аннотации означают «генератор, который отдает объекты типа `T`, но не потребляет и не возвращает значений». Генераторы, способные потреблять и возвращать значения, называются сопрограммами, это наша следующая тема.

Классические сопрограммы

В документе PEP 342 «Coroutines via Enhanced Generators» (*https://peps.python.org/pep-0342/*) был введен метод `.send()` и другие средства, благодаря которым стало возможно использовать генераторы как сопрограммы. В PEP 342 слово «сопрограмма» употребляется в том же смысле, что в этом разделе.

К сожалению, в официальной документации по Python и в стандартной библиотеке теперь употребляется несогласованная терминология в отношении генераторов, используемых в качестве сопрограмм, что заставляет меня применять термин «классиче-

[1] В версии 0.910 Муру все еще использует объявленные нерекомендуемыми типы из модуля `typing`.

> ская сопрограмма» в противоположность появившимся позднее объектам «платформенных сопрограмм».
>
> После выхода версии Python 3.5 наметилась тенденция использовать слово «сопрограмма» как синоним «платформенной сопрограммы». Но документ PEP 342 не объявлен нерекомендуемым, и классические сопрограммы по-прежнему работают, как было задумано, хотя уже и не поддерживаются пакетом `asyncio`.

Понять классические сопрограммы в Python мудрено, потому что на самом деле это генераторы, только используются по-другому. Поэтому вернемся на шаг назад и рассмотрим еще одно средство Python, которое можно использовать двумя способами.

В разделе «Кортеж – не просто неизменяемый список» главы 2 мы видели, что экземпляры `tuple` можно использовать как записи или как неизменяемые последовательности. Если кортеж используется как запись, то ожидается, что в нем будет определенное число элементов и у каждого элемента будет свой тип. При использовании в качестве неизменяемого списка кортеж может иметь произвольную длину, но ожидается, что все элементы будут иметь один и тот же тип. Поэтому есть два разных способа аннотирования кортежей типами:

```
# Запись о городе, содержащая название, страну и численность населения:
city: tuple[str, str, int]

# Неизменяемая последовательность доменов:
domains: tuple[str, ...]
```

Нечто подобное происходит и с генераторами. Обычно они используются как итераторы, но могут использоваться и как сопрограммы. *Сопрограмма* – это в действительности генераторная функция, в теле которой имеется ключевое слово `yield`. А *объект сопрограммы* физически является объектом-генератором. Несмотря на общую реализацию на C, использование генераторов и сопрограмм в Python настолько отличается, что существует два способа их аннотирования:

```
# Переменную `readings` можно связать с объектом итератора или генератора,
# отдающим элементы типа `float`:
readings: Iterator[float]

# Переменную `sim_taxi` можно связать с сопрограммой, которая представляет
# такси в алгоритме моделирования дискретных событий. Она отдает события,
# получает временные метки типа `float` и возвращает количество поездок
# за время моделирования:
sim_taxi: Generator[Event, float, int]
```

Как будто этого мало, чтобы внести путаницу, авторы модуля `typing` решили назвать этот тип `Generator`, хотя на самом деле он описывает API объекта-генератора, который предполагается использовать как сопрограмму, тогда как генераторы чаще используются в роли простых итераторов.

В документации по модулю `typing` (*https://docs.python.org/3/library/typing.html#typing.Generator*) формальные параметры-типы `Generator` описаны следующим образом:

```
Generator[YieldType, SendType, ReturnType]
```

`SendType` нужен, только когда генератор используется в качестве сопрограммы. Этот параметр описывает тип `x` в вызове `gen.send(x)`. Ошибкой является вызов `.send()` от имени генератора, который предполагалось использовать в качестве итератора, а не сопрограммы. Аналогично параметр-тип `ReturnType` имеет смысл только для аннотирования сопрограммы, потому что итераторы не возвращают значения так, как делают обычные функции. Единственное разумное действие с генератором, используемым в роли итератора, – вызов метода `next(it)` прямо или косвенно посредством цикла `for` и других форм итерирования. `YieldType` – тип значения, возвращаемого при обращении к `next(it)`.

Тип `Generator` имеет такие же параметры-типы, как `typing.Coroutine` (*https://docs.python.org/3.10/library/typing.html#typing.Coroutine*):

```
Coroutine[YieldType, SendType, ReturnType]
```

В документации по `typing.Coroutine` написано: «Вариантность и порядок переменных-типов такой же, как в типе `Generator`». Но типы `typing.Coroutine` (объявленный нерекомендуемым) и `collections.abc.Coroutine` (обобщенный начиная с версии Python 3.9) предназначены для аннотирования только платформенных, но не классических сопрограмм. Если вы хотите использовать аннотации типов в сочетании с классическими сопрограммами, то будете испытывать замешательство, аннотируя их как `Generator[YieldType, SendType, ReturnType]`.

Дэвид Бизли посвятил классическим сопрограммам некоторые из своих лучших и наиболее полных презентаций. В информационных материалах, которые раздавались на конференции PyCon 2009 (*http://www.dabeaz.com/coroutines/Coroutines.pdf*), есть такие положения:

- генераторы порождают данные для итерирования;
- сопрограммы являются потребителями данных;
- если не хотите, чтобы сорвало крышу, не путайте эти две концепции;
- сопрограммы не имеют никакого отношения к итерированию;
- примечание: у применения `yield` для порождения значения в сопрограмме есть свои резоны, но с итерированием они не связаны[1].

Теперь посмотрим, как работают классические сопрограммы.

Пример: сопрограмма для вычисления накопительного среднего

При обсуждении замыканий в главе 9 мы рассматривали объект для вычисления накопительного среднего: в примере 9.7 приведен простой класс, а в примере 9.13 – функция высшего порядка, порождающая замыкание для запоминания переменных `total` и `count` между вызовами. В примере 17.37 показано, как то же самое сделать с помощью сопрограммы[2].

[1] Слайд 33 «Keeping It Straight» из презентации «A Curious Course on Coroutines and Concurrency» (*http://www.dabeaz.com/coroutines/Coroutines.pdf*).

[2] В основу этого примера положен фрагмент, приведенный Джекобом Холмом в списке рассылки Python-ideas, его сообщение называется «Yield-From: Finalization guarantees» (*https://mail.python.org/pipermail/python-ideas/2009-April/003841.html*). Позже в той же ветке появились вариации на эту тему, а сам Холм объяснил ход своих мыслей в сообщении 003912 (*https://mail.python.org/pipermail/python-ideas/2009-April/003912.html*).

Пример 17.37. coroaverager0.py: сопрограмма для вычисления накопительного среднего

```
from collections.abc import Generator

def averager() -> Generator[float, float, None]: ❶
    total = 0.0
    count = 0
    average = 0.0
    while True: ❷
        term = yield average ❸
        total += term
        count += 1
        average = total / count
```

❶ Эта функция возвращает генератор, который отдает значения типа `float`, принимает значения с помощью метода `.send()` и не возвращает никакого полезного значения[1].

❷ В этом бесконечном цикле сопрограмма будет отдавать средние, пока клиентский код посылает значения.

❸ Здесь предложение `yield` используется, чтобы приостановить сопрограмму, отдать результат вызывающей стороне и – впоследствии – получить значение, посланное вызывающей стороной, после чего выполнение бесконечного цикла продолжится.

В сопрограмме `total` и `count` могут быть локальными переменными: для запоминания контекста на время, пока сопрограмма приостановлена в ожидании следующего вызова `.send()`, не нужны ни атрибуты экземпляра, ни замыкания. Потому-то сопрограммы и являются привлекательной альтернативой обратным вызовам при асинхронном программировании – они сохраняют локальное состояние между активациями.

В примере 17.38 приведены тесты, демонстрирующие использование сопрограммы `averager`.

Пример 17.38. coroaverager0.py: тест, демонстрирующий использование сопрограммы вычисления накопительного среднего из примера 17.37

```
>>> coro_avg = averager() ❶
>>> next(coro_avg) ❷
0.0
>>> coro_avg.send(10) ❸
10.0
>>> coro_avg.send(30)
20.0
>>> coro_avg.send(5)
15.0
```

❶ Создать объект сопрограммы.

❷ Запустить сопрограмму. При этом отдается начальное значение `average`: 0.0.

❸ Теперь мы в деле: каждый вызов `.send()` отдает текущее среднее.

[1] На самом деле он вообще не возвращает управления, если только какое-то исключение не приведет к выходу из цикла. Муру 0.910 принимает как `None`, так и `typing.NoReturn` в качестве типа значения, возвращаемого генератором, но она также принимает в этой позиции `str`, т. е., похоже, еще не умеет в полной мере анализировать код сопрограмм.

В этом тесте вызов `next(coro_avg)` заставляет сопрограмму дойти до `yield`, при этом будет отдано начальное значение `average`. Запустить сопрограмму можно также, вызвав `coro_avg.send(None)`, – именно так и поступает встроенная функция `next()`. Но отправить какое-то значение, кроме `None`, нельзя, потому что сопрограмма может принимать отправленные значения, только когда приостановлена в точке `yield`. Вызов `next()` или `.send(None)`, чтобы продвинуть выполнение к первому предложению `yield`, называется «инициализацией сопрограммы».

После каждой активации сопрограмма приостанавливается на выражении `yield` и ждет отправки значения. В строке `coro_avg.send(10)` значение отправляется, после чего сопрограмма активируется. Выражение `yield` отдает значение 10, которое присваивается переменной `term`. В оставшейся части цикла обновляются переменные `total`, `count` и `average`. На следующей итерации цикла `while` отдается значение `average`, и сопрограмма снова приостанавливается в точке `yield`.

У внимательного читателя, наверное, возник вопрос, как остановить работу объекта `averager` (`coro_avg`), – ведь цикл-то бесконечный. Обычно нам не нужно завершать генератор, потому что сборщик мусора позаботится о нем, как только на него не останется ни одной ссылки. Если все-таки необходимо завершить генератор явно, воспользуйтесь методом `.close()`, как показано в примере 17.39.

Пример 17.39. coroaverager.py: продолжение примера 17.38

```
>>> coro_avg.send(20) ❶
16.25
>>> coro_avg.close() ❷
>>> coro_avg.close() ❸
>>> coro_avg.send(5) ❹
Traceback (most recent call last):
  ...
StopIteration
```

❶ `coro_avg` – экземпляр, созданный в примере 17.38.

❷ Метод `.close()` возбуждает исключение `GeneratorExit` в приостановленном выражении `yield`. Не будучи обработано в функции сопрограммы, исключение завершает ее, а затем перехватывается объектом-генератором, обертывающим сопрограмму, – потому-то мы его и не видим.

❸ Вызов `.close()` для уже закрытой сопрограммы ничего не делает.

❹ Попытка выполнить `.send()` для закрытой сопрограммы возбуждает исключение `StopIteration`.

Помимо метода `.send()`, в документе PEP 342 «Coroutines via Enhanced Generators» (*https://peps.python.org/pep-0342/*) описан также способ возврата значения из сопрограммы. В следующем разделе показано, как это делается.

Возврат значения из сопрограммы

Теперь мы изучим еще одну сопрограмму для вычисления среднего. Эта версия не отдает частичные результаты, а возвращает кортеж, содержащий количество членов последовательности и их среднее. Я разбил код на две части: примеры 17.40 и 17.41.

Пример 17.40. coroaverager2.py: начало файла

```
from collections.abc import Generator
from typing import Union, NamedTuple

class Result(NamedTuple): ❶
    count: int # type: ignore ❷
    average: float

class Sentinel: ❸
    def __repr__(self):
        return f'<Sentinel>'

STOP = Sentinel() ❹

SendType = Union[float, Sentinel] ❺
```

❶ Сопрограмма `averager2` в примере 17.41 вернет экземпляр `Result`.

❷ `Result` – это подкласс `tuple`, имеющий метод `.count()`, который мне не нужен. Комментарий `# type: ignore` подавляет сообщения Муру на предмет наличия поля `count`[1].

❸ Этот класс нужен только для того, чтобы сделать сигнальное значение понятным с помощью метода `__repr__`.

❹ Сигнальное значение, которое я буду использовать, чтобы заставить сопрограмму прекратить сбор данных и вернуть результат.

❺ Этот псевдоним типа я буду использовать в качестве второго параметра-типа в типе, возвращаемом сопрограммой `Generator`.

Это определение `SendType` работает и в версии Python 3.10, но если вам не нужно поддерживать более ранние версии, то лучше записать его следующим образом, предварительно импортировав `TypeAlias` из `typing`:

```
SendType: TypeAlias = float | Sentinel
```

Использование `|` вместо `typing.Union` выглядит так коротко и понятно, что я бы, пожалуй, предпочел вообще не создавать этот псевдоним типа, а записать сигнатуру `averager2`, как показано ниже:

```
def averager2(verbose: bool=False) -> Generator[None, float | Sentinel, Result]:
```

Теперь рассмотрим код самой сопрограммы.

Пример 17.41. coroaverager2.py: сопрограмма, которая возвращает результирующее значение

```
def averager2(verbose: bool = False) -> Generator[None, SendType, Result]: ❶
    total = 0.0
    count = 0
    average = 0.0
    while True:
        term = yield ❷
```

[1] Я подумывал переименовать это поле, но `count` – лучшее имя для локальной переменной в сопрограмме, и именно это имя и использовал для переменной в аналогичных примерах, так что имеет смысл оставить его для поля `Result`. Я без колебаний пишу `# type: ignore`, чтобы не натыкаться на ограничения и помехи со стороны средств статической проверки типов и не делать код хуже или сложнее, только чтобы порадовать инструмент.

```
        if verbose:
            print('received:', term)
        if isinstance(term, Sentinel): ❸
            break
        total += term ❹
        count += 1
        average = total / count
    return Result(count, average) ❺
```

❶ В этой сопрограмме тип отдаваемого значения `None`, потому что она не отдает никаких данных. Она получает данные типа `SendType` и возвращает кортеж типа `Result` по завершении.

❷ Подобное использование `yield` имеет смысл только в сопрограммах, предназначенных для потребления данных. Здесь `yield` отдает `None`, но получает `term` от `.send(term)`.

❸ Если `term` является экземпляром `Sentinel`, то выйти из цикла. Благодаря этой проверке с помощью `isinstance`...

❹ ...Муру позволяет прибавить `term` к `total`, не надоедая мне жалобами на то, что нельзя прибавлять `float` к объекту, который может иметь тип `float` или `Sentinel`.

❺ До этой строки дело дойдет, только если сопрограмме отправлен экземпляр `Sentinel`.

Теперь посмотрим, как использовать эту сопрограмму, и начнем с простого примера, который не порождает никакого результата.

Пример 17.42. coroaverager2.py: тест, демонстрирующий использование `.cancel()`

```
>>> coro_avg = averager2()
>>> next(coro_avg)
>>> coro_avg.send(10) ❶
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> coro_avg.close() ❷
```

❶ Напомним, что `averager2` не отдает частичных результатов. А отдает она значение `None`, которое консоль Python не отображает.

❷ Вызов `.close()` в этой сопрограмме заставляет ее прекратить выполнение, но не вернуть результат, потому что в строке `yield` возбуждается исключение `GeneratorExit` и до предложения `return` поток выполнения не доходит.

Теперь заставим сопрограмму работать, см. пример 17.43.

Пример 17.43. coroaverager2.py: тест, демонстрирующий `StopIteration` с возвратом `Result`

```
>>> coro_avg = averager2()
>>> next(coro_avg)
>>> coro_avg.send(10)
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> try:
...     coro_avg.send(STOP) ❶
... except StopIteration as exc:
```

```
...     result = exc.value ❷
...
>>> result ❸
Result(count=3, average=15.5)
```

❶ Отправка сигнального значения `STOP` заставляет сопрограмму выйти из цикла и вернуть `Result`. Объект-генератор, обертывающий сопрограмму, затем возбуждает исключение `StopIteration`.
❷ В экземпляре `StopIteration` атрибут `value` связан со значением предложения `return`, завершившего сопрограмму.
❸ Хотите верьте, хотите нет!

Эта идея «контрабандой протащить» возвращенное значение из сопрограммы, обернутой исключением `StopIteration`, кажется нечестным приемом. Тем не менее он описан в документе PEP 342 «Coroutines via Enhanced Generators» и документирован в описании исключения `StopIteration` (*https://docs.python.org/3/reference/expressions.html #yield-expressions*) и в разделе «Выражения yield» (*https://docs.python.org/3/reference/expressions.html#yield-expressions*) главы 6 справочного руководства по языку Python.

Делегирующий генератор может получить возвращенное сопрограммой значение непосредственно, воспользовавшись конструкцией `yield from`, как показано в примере 17.44.

Пример 17.44. coroaverager2.py: тест, демонстрирующий `StopIteration` с возвратом `Result`

```
>>> def compute():
...     res = yield from averager2(True) ❶
...     print('computed:', res) ❷
...     return res ❸
...
>>> comp = compute() ❹
>>> for v in [None, 10, 20, 30, STOP]: ❺
...     try:
...         comp.send(v) ❻
...     except StopIteration as exc: ❼
...         result = exc.value
received: 10
received: 20
received: 30
received: <Sentinel>
computed: Result(count=3, average=20.0)
>>> result ❽
Result(count=3, average=20.0)
```

❶ В `res` попадает значение, возвращенное `averager2`; механизм `yield from` извлекает возвращенное значение при обработке исключения `StopIteration`, знаменующего завершение сопрограммы. Если параметр `verbose` равен `True`, то сопрограмма печатает полученное значение, чтобы было видно, как она работает.
❷ Следите за тем, что выводит эта строка во время работы генератора.
❸ Вернуть результат. Он тоже будет обернут экземпляром `StopIteration`.

❹ Создать объект делегирующей сопрограммы.
❺ Этот цикл управляет делегирующей сопрограммой.
❻ Сначала посылается значение `None`, чтобы инициализировать сопрограмму, а в конце сигнальное значение, чтобы остановить ее.
❼ Перехватить `StopIteration`, чтобы извлечь из него значение, возвращенное `compute`.
❽ После строк, напечатанных `averager2` и `compute`, мы получаем экземпляр `Result`.

Хотя эти примеры не делают ничего особенного, следить за кодом трудно. Управлять сопрограммой с помощью вызовов `.send()` и извлечения результата сложно, если не прибегать к `yield from`, но этот синтаксис допустим только внутри делегирующего генератора или сопрограммы, которые в конечном итоге должны приводиться в действие нетривиальным кодом типа показанного в примере 17.44.

Из предыдущих примеров видно, что прямое использование сопрограмм утомительно и запутанно. А если добавить обработку исключений и метод сопрограмм `.throw()`, то примеры станут еще более мудреными. Я не буду рассматривать метод `.throw()` в этой книге, потому что он, как и `.send()`, полезен только для управления сопрограммами «вручную», чего я делать не рекомендую, если только вы не занимаетесь разработкой нового основанного на сопрограммах каркаса с нуля.

Если вас интересует углубленное рассмотрение классических сопрограмм, включая и метод `.throw()`, обратитесь к статье «Классические сопрограммы» (*https://www.fluentpython.com/extra/classic-coroutines/*) на сопроводительном сайте. Там имеется псевдокод на языке, похожем на Python, показывающий, как `yield from` приводит в действие генераторы и сопрограммы, а также небольшая программа моделирования дискретных событий, демонстрирующая форму конкурентности с использованием сопрограмм, но без применения каркаса асинхронного программирования.

На практике для продуктивной работы с сопрограммами необходима поддержка со стороны специализированного каркаса. Именно это и предоставлял модуль `asyncio` для классических сопрограмм в Python 3.3. После включения платформенных сопрограмм в Python 3.5 разработчики ядра Python постепенно прекращают поддержку классических сопрограмм в `asyncio`. Но базовые механизмы очень похожи. Синтаксическая конструкция `async def` позволяет легко находить платформенные сопрограммы в коде, что само по себе является важным преимуществом. Внутри платформенной сопрограммы используется `await` вместо `yield from` для делегирования работы другим сопрограммам. Всему этому посвящена глава 21.

Завершим эту главу головоломным разделом, посвященным ковариантности и контравариантности в аннотациях типов для сопрограмм.

Аннотации обобщенных типов для классических сопрограмм

В разделе «Контравариантные типы» главы 15 я упоминал `typing.Generator` как один из немногих стандартных библиотечных типов с контравариантным параметром-типом. Теперь, изучив классические сопрограммы, мы готовы разобраться с этим обобщенным типом.

Вот как был объявлен тип `typing.Generator` в модуле *typing.py* для версии Python 3.6[1]:

```
T_co = TypeVar('T_co', covariant=True)
V_co = TypeVar('V_co', covariant=True)
T_contra = TypeVar('T_contra', contravariant=True)

# много строк опущено

class Generator(Iterator[T_co], Generic[T_co, T_contra, V_co],
                extra=_G_base):
```

Это объявление обобщенного типа означает, что аннотация типа `Generator` требует трех параметров-типов, которые мы уже видели раньше:

```
my_coro : Generator[YieldType, SendType, ReturnType]
```

Из описания переменных-типов в формальных параметрах видно, что типы `YieldType` и `ReturnType` ковариантны, но `SendType` контравариантен. Чтобы понять, почему это так, примите во внимание, что `YieldType` и `ReturnType` – «выходные» типы. Оба описывают данные, исходящие из объекта сопрограммы, т. е. когда объект-генератор используется в роли сопрограммы.

Ковариантность относительно этих параметров имеет смысл, потому что любой код, ожидающий сопрограмму, которая отдает числа с плавающей точкой, может работать с сопрограммой, отдающей целые. Именно поэтому тип `Generator` ковариантен относительно параметра `YieldType`. Аналогичное рассуждение применимо к параметру `ReturnType`, тоже ковариантному.

Применяя нотацию, введенную в разделе «Ковариантные типы» главы 15, мы можем выразить ковариантность первого и третьего параметров с помощью символов `:>`, указывающих в одном направлении:

```
                          float :> int
Generator[float, Any, float] :> Generator[int, Any, int]
```

`YieldType` и `ReturnType` – примеры применения первого правила из раздела «Эвристические правила вариантности» главы 15.

1. Если формальный параметр-тип определяет тип данных, исходящих из объекта, то он может быть ковариантным.

С другой стороны, `SendType` – «входной» параметр: это тип аргумента `value` в методе `.send(value)` объекта сопрограммы. Клиентский код должен отправлять сопрограмме числа с плавающей точкой, не может работать с сопрограммой, принимающей `int` в качестве `SendType`, потому что `float` не является подтипом `int`. Иными словами, `float` не *совместим с* `int`. Но клиент может работать с сопрограммой, для которой в роли `SendType` выступает `complex`, потому что `float` является подтипом `complex`, т. е. `float` *совместим с* `complex`.

Нотация `:>` делает контравариантность второго параметра видимой:

[1] Начиная с версии Python 3.7 `typing.Generator` и другие типы, соответствующие ABC в модуле `collections.abc`, были переработаны и теперь являются оберткой вокруг соответствующего ABC, так что их обобщенные параметры не видны в исходном файле `typing.py`. Поэтому я здесь ссылаюсь на исходный код Python 3.6.

```
                    float :> int
Generator[Any, float, Any] <: Generator[Any, int, Any]
```

Это пример применения второго эвристического правила вариантности.

2. Если формальный параметр-тип определяет тип данных, входящих в объект после его начального конструирования, то он может быть контравариантным.

На этом веселом обсуждении вариантности мы завершаем самую длинную главу данной книги.

Резюме

Итерирование так глубоко укоренилось в языке, что я часто говорю, что Python пропитан итераторами[1]. Интеграция паттерна Итератор в семантику Python – яркий пример того, что паттерны проектирования не в одинаковой степени применимы во всех языках. В Python классический итератор, реализованный «вручную», как в примере 17.4, не имеет никакой практической ценности, а разве что педагогическую.

В этой главе мы написали несколько вариантов класса для обхода слов в текстовом файле, возможно, очень длинном. Мы видели, как Python использует встроенную функцию `iter()` для создания итераторов из объектов, похожих на последовательности. Мы построили классический итератор как класс с методом `__next__()`, а затем использовали генераторы, чтобы сделать каждую последующую версию класса `Sentence` короче и понятнее.

Затем мы написали генератор арифметических прогрессий и показали, как с помощью модуля `itertools` упростить его. Далее познакомились с большинством генераторных функций общего назначения из стандартной библиотеки.

Потом изучили выражения `yield from` на примерах простых генераторов `chain` и `tree`.

Последний крупный раздел был посвящен классическим сопрограммам, хотя эта тема постепенно сходит на нет после добавления платформенных сопрограмм в версию Python 3.5. Классические сопрограммы, хотя их трудно использовать на практике, являются основой платформенных сопрограмм, а выражение `yield from` – прямой предок `await`.

Также были рассмотрены аннотации для типов `Iterable`, `Iterator` и `Generator`, последний из которых дает конкретный и редкий пример контравариантного параметра-типа.

Дополнительная литература

Детальное техническое описание генераторов можно найти в разделе 6.2.9 «Выражения yield» справочного руководства по языку Python (*https://docs.python.org/3/reference/expressions.html#yieldexpr*). Генераторные функции были

[1] В оригинале употреблено слово «grok» и приводится такое пояснение: согласно справочнику жаргона (*http://catb.org/~esr/jargon/html/G/grok.html*), *grok* означает не просто «выучить что-то», а впитать, так что «это становится частью тебя, твоей личности».

впервые определены в документе PEP 255 «Simple Generators» (*https://www.python.org/dev/peps/pep-0255/*).

Документация по модулю `itertools` (*https://docs.python.org/3/library/itertools.html*) – отличный источник информации благодаря включенным примерам. Хотя функции из этого модуля написаны на C, в документации показано, что многие из них можно было бы реализовать и на Python, часто с привлечением других функций из того же модуля. Примеры подобраны замечательно; например, в одном фрагменте показано, как с помощью функции `accumulate` погасить ссуду с процентами, если задан график платежей. А в разделе «Рецепты itertools» (*https://docs.python.org/3/library/ itertools.html#itertools-recipes*) описаны дополнительные высокопроизводительные функции, построенные на базе функций из `itertools`.

Помимо стандартной библиотеки Python, я рекомендую пакет More Itertools (*https://more-itertools.readthedocs.io/en/stable/index.html*), который, следуя традиции `itertools`, предоставляет мощные генераторы с многочисленными примерами и рядом полезных рецептов.

В главе 4 «Итераторы и генераторы» книги David Beazley, Brian K. Jones «*Python Cookbook*», 3-издание (O'Reilly), приведено 16 рецептов, где эта тема рассматривается с разных точек зрения, но всегда с прицелом на практическое применение. Имеются поучительные рецепты с использованием `yield from`.

Себастьян Риттау, в настоящее время руководитель группы сопровождения *typeshed*, объясняет, почему итераторы должны быть итерируемыми объектами, в статье 2006 года «Java: Iterators are not Iterable».

Синтаксическая конструкция `yield from` объясняется на примерах в документе «What's New in Python 3.3» (см. PEP 380 «Syntax for Delegating to a Subgenerator», *https://docs.python.org/3/whatsnew/3.3.html#pep-380-syntax-for-delegating-to-a-subgenerator*). В моей статье «Классические сопрограммы» (*https://www.fluentpython.com/extra/classic-coroutines/*) на сопроводительном сайте книги имеется углубленное рассмотрение `yield from`, включающее псевдокод его реализации на C.

Дэвид Бизли – непререкаемый авторитет во всем, что касается генераторов и сопрограмм в Python. В книге «*Python Cookbook*», 3-е издание, (O'Reilly), написанной в соавторстве с Брайаном Джонсом, приведены многочисленные примеры сопрограмм. Пособия Бизли на эту тему, представленные на конференции PyCon, хорошо известны глубиной и широтой охвата. Первое было представлено на PyCon US 2008: «Generator Tricks for Systems Programmers» (*http://www.dabeaz.com/generators/*). На PyCon US 2009 мы увидели легендарное «A Curious Course on Coroutines and Concurrency» (*http://www.dabeaz.com/coroutines/*) (привожу ссылки на все три части видео, которые трудно найти в сети: *https://archive.org/details/pyvideo_213___pycon-2009-a-curious-course-on-coroutines-and-concurrency-part-1-of-3*, *https://archive.org/details/pyvideo_215___pycon-2009-a-curious-course-on-coroutines-and-concurrency-part-2-of-3*, *http://www.dabeaz.com/finalgenerator/*). На конференции PyCon 2014 в Монреале он представил пособие «Generators: The Final Frontier» (*http://www.dabeaz.com/finalgenerator/*), в котором привел дополнительные примеры, относящиеся к конкурентности, теме главы 21. Дэйв не может удержаться от искушения взорвать мозги своим слушателям, поэтому в последней части презентации «The Final Frontier» классический паттерн Посетитель в вычислителе арифметических выражений был заменен сопрограммами.

Сопрограммы предлагают новые способы организации кода; чтобы освоить их

возможности, нужно время – но не так ли обстоит дело с рекурсией и полиморфизмом (динамической диспетчеризацией)? Интересный пример классического алгоритма, переписанного с помощью сопрограмм, приведен в статье Джеймса Пауэлла «Greedy algorithm with coroutines» (*https://web.archive.org/web/20200218150637/http://seriously.dontusethiscode.com/2013/05/01/greedy-coroutine.html*).

В книге Brett Slatkins «Effective Python», 1-е издание (Addison-Wesley), есть великолепная главка «Рассматривайте сопрограммы как способ выполнить много функций одновременно». Она не включена во второе издание, но осталась доступна в сети как пример главы (*https://effectivepython.com/2015/03/10/consider-coroutines-to-run-many-functions-concurrently*). Слаткин представляет лучший из известных мне примеров управления сопрограммами с помощью `yield from`: реализацию игры «Жизнь» (*https://en.wikipedia.org/wiki/Conway's_Game_of_Life*) Джона Конвея, в которой сопрограммы отвечают за состояние каждой клетки в процессе игры. Я переработал код этого примера – выделил функции и классы из тестовых фрагментов, составляющих оригинальный код Слаткина. Я также переписал тесты в формате doc-тестов, чтобы результат различных сопрограмм и классов можно было видеть, не запуская скрипт. Переработанный пример размещен на GitHub (*https://gist.github.com/ramalho/da5590bc38c973408839*).

Поговорим

Минималистский интерфейс итератора в Python

В разделе «Реализация» главы о паттерне Итератор книги «Банды четырех» написано:

> Минимальный интерфейс класса Iterator состоит из операций First, Next, IsDone и CurrentItem.

Однако к этому предложению относится такая сноска:

> Этот интерфейс можно и еще уменьшить, если объединить операции Next, IsDone и CurrentItem в одну, которая будет переходить к следующему объекту и возвращать его. Если обход завершен, то эта операция вернет специальное значение (например, 0), обозначающее конец итерации.

Это близко к тому, что мы имеем в Python: всю работу делает один метод `__next__`. Но вместо специального значения, на которое по ошибке можно не обратить внимания, о конце итерации возвещает исключение `StopIteration`. Просто и правильно: таков путь Python.

Взаимозаменяемые генераторы

Всякий занимающийся большими наборами данных найдет много применений генераторам. Расскажу о том, как я в первый раз создавал практическое решение на основе генераторов.

Много лет назад я работал в BIREME, цифровой библиотеке под управлением PAHO/WHO (Панамериканская организация здравоохранения / Всемирная организация здравоохранения) в Сан-Паулу, Бразилия. В числе прочих BIREME создала библиографические наборы данных LILACS (указатель по наукам о здравоохранении в Латинской Америке и странах Карибского бассейна) и SciELO (онлай-

новая научная электронная библиотека). Это две весьма полные базы данных по региональной научно-исследовательской литературе в области здравоохранения.

Начиная с конца 1980-х годов для управления LILACS использовалась нереляционная документная база данных CDS/ISIS, созданная ЮНЕСКО. Моей задачей было, в частности, изучить возможные альтернативы для миграции LILACS, а затем и гораздо более объемной SciELO в современную базу данных с открытым исходным кодом типа CouchDB или MongoDB. В то время я написал статью, в которой рассматривал слабоструктурированную модель данных и различные способы представления данных из CDS/ISIS в формате записей JSON: «From ISIS to CouchDB: Databases and Data Models for Bibliographic Records» (*https://journal.code4lib.org/articles/4893*).

Частью этого исследования стал Python-скрипт для чтения файла CDS/ISIS и записи JSON-файла, пригодного для импорта в CouchDB или MongoDB. Сначала скрипт читал файлы в формате ISO-2709, экспортированные из CDS/ISIS. Чтение и запись нужно было производить инкрементно, потому что полные наборы данных были гораздо больше доступной оперативной памяти. С этим проблем не возникло: на каждой итерации главного цикла `for` читалась одна запись из *iso*-файла, обрабатывалась и записывалась в *json*-файл.

Однако, по эксплуатационным соображениям, нужно было, чтобы скрипт *isis2json.py* поддерживал еще один формат данных CDS/ISIS: двоичные *mst*-файлы, которые использовались в BIREME, – чтобы избежать дорогостоящего экспорта в формат ISO-2709. Вот теперь возникла проблема: у библиотек, применявшихся для чтения файлов ISO-2709 и *mst*-файлов, был совершенно различный API. А цикл записи JSON-файла и так уже был достаточно сложным, поскольку скрипт принимал много параметров, управляющих реструктуризацией записей. Чтение данных с помощью двух разных API в одном цикле `for`, порождавшем JSON-файл, сделало бы программу слишком громоздкой.

Решение было найдено: изолировать логику чтения в двух генераторных функциях, по одной для каждого поддерживаемого формата. В итоге я разбил скрипт *isis2json.py* на четыре функции. Исходный код на Python 2 вместе с зависимостями есть в репозитории *fluentpython/isis2json* на GitHub (*https://github.com/fluentpython/isis2json*)[1].

Ниже описана высокоуровневая структура скрипта:

`main`
: Функция `main` вызывает `argparse`, чтобы прочитать аргументы командной строки, настраивающие структуру выходных записей. На основе расширения имени входного файла выбирается подходящая генераторная функция чтения данных и отдачи записей, по одной.

`iter_iso_records`
: Эта генераторная функция читает *iso*-файлы (в предположении, что они записаны в формате ISO-2709). Она принимает два аргумента: имя файла и `isis_json_type`, один из флагов, относящихся к структуре записи. На каждой итерации цикла for читается одна запись, создается пустой словарь `dict`, этот словарь заполняется данными полей и отдается.

[1] Код написан на Python 2, потому что одной из факультативных зависимостей стала библиотека на Java под названием *Bruma*, которую можно импортировать, когда скрипт работает под управлением Jython, а он еще не поддерживает Python 3.

`iter_mst_records`
: Эта генераторная функция читает *mst*-файлы[1]. Заглянув в исходный код *isis2json.py*, вы увидите, что она посложнее `iter_iso_records`, но интерфейс и общая структура такие же: функция принимает имя файла и аргумент `isis_json_type`, после чего входит в цикл `for`, где на каждой итерации строится и отдается словарь, представляющий одну запись.

`write_json`
: Эта функция выводит JSON-записи, по одной за раз. У нее много аргументов, но первым является `input_gen` – ссылка на генераторную функцию: `iter_iso_records` или `iter_mst_records`. Главный цикл `for` в функции `write_json` перебирает словари, отданные выбранным генератором, реструктурирует их в зависимости от аргументов командной строки и добавляет JSON-запись в конец выходного файла.

Благодаря генераторным функциям я смог отделить чтение от записи. Конечно, проще всего было прочитать все записи в память, а затем записать их на диск. Но из-за размера наборов данных такой путь не годился. Генераторы позволили чередовать чтение и запись, поэтому скрипт мог обрабатывать файлы любого размера. Кроме того, специальная логика чтения записи в разных форматах ввода отделена от логики реструктуризации записей для вывода.

Теперь если понадобится использовать *isis2json.py* для поддержки дополнительного формата ввода, например MARCXML, DTD-схемы, используемой в Библиотеке Конгресса США для представления данных в формате ISO-2709, то легко будет написать третью генераторную функцию для реализации логики чтения. А в сложной функции `write_json` ничего изменять не придется.

Это, конечно, не высшая математика, но реальный пример ситуации, в которой генераторы позволили получить эффективное и гибкое решение для обработки базы данных в виде потока записей. При этом потребление памяти остается низким вне зависимости от размера набора данных.

[1] Библиотека для чтения сложных двоичных *mst*-файлов написана на Java, так что эта функциональность доступна, только когда скрипт *isis2json.py* выполняется интерпретатором Jython версии 2.5 или выше. Детали см. в файле *README.rst* в репозитории. Зависимости импортируются внутри нуждающихся в них генераторных функций, поэтому скрипт может работать даже тогда, когда доступна только одна из внешних библиотек.

Глава 18

Блоки with, match и else

> Не исключено, что контекстные менеджеры окажутся почти такими же важными, как сами подпрограммы. Мы затронули лишь самую верхушку айсберга [...]. В языке Basic есть предложение `with`, как и во многих других языках. Но все они делают совсем не то – они лишь экономят время на повторяющемся поиске атрибутов с точкой, не производя ни инициализации, ни очистки. Не нужно думать, что раз названия одинаковы, то одинаковы и функции. Предложение `with` – очень мощная штука.
>
> – Раймонд Хэттингер, страстный проповедник Python[1]

В этой главе мы обсудим средства управления потоком выполнения, которые не так часто встречаются в других языках и потому остаются малоизвестными программистам на Python или используются ими недостаточно эффективно. Вот эти средства:

- предложение `with` и протокол контекстных менеджеров;
- сопоставление с образцом с помощью конструкции `match/case`;
- часть `else` в предложениях `for`, `while` и `try`.

Предложение `with` организует временный контекст и гарантированно очищает его под контролем объекта контекстного менеджера. Это позволяет предотвратить ошибки и уменьшить объем стереотипного кода, одновременно сделав API безопаснее и проще в использовании. Программисты на Python находят много применений блокам `with` помимо автоматического закрытия файлов.

Мы уже видели сопоставление с образцом в других главах, а здесь посмотрим, как можно выразить грамматику языка в виде последовательностей-образцов. Это позволит понять, почему `match/case` является эффективным инструментом для создания языковых процессоров, понятных и легко расширяемых. Мы рассмотрим полный интерпретатор для небольшого, но функционального подмножества языка Scheme. Те же идеи можно применить к разработке языка шаблонов или предметно-ориентированного языка (DSL) в более крупной системе.

Предложение `else` не представляет никаких сложностей, но при правильном использовании в сочетании с `for`, `while` и `try` позволяет программисту лучше выразить намерения.

---

[1] Тезисы доклада на конференции PyCon US 2013 «What Makes Python Awesome» (*http://pyvideo.org/video/1669/keynote-3*); часть, относящаяся к `with`, начинается в 23:00 и заканчивается в 26:15.

Что нового в этой главе

Раздел «Сопоставление с образцом в lis.py: развернутый пример» новый.

Я переработал раздел «Утилиты contextlib», включив в него новые средства из модуля `contextlib`, добавленные начиная с версии Python 3.6, и новый скобочный синтаксис контекстных менеджеров, появившийся в Python 3.10.

Начнем с мощного предложения `with`.

Контекстные менеджеры и блоки with

Объекты контекстных менеджеров служат для управления предложением `with`, точно так же, как итераторы управляют предложением `for`.

Предложение `with` было задумано для того, чтобы упростить конструкцию `try/finally`, гарантирующую, что некоторая операция будет выполнена после блока, даже если этот блок прерван в результате исключения, предложения `return` или вызова `sys.exit()`. Код внутри части `finally` обычно освобождает критически важный ресурс или восстанавливает временно измененное состояние.

Сообщество Python находит новые необычные применения для контекстных менеджеров. Вот несколько примеров из стандартной библиотеки:

- управление транзакциями в модуле `sqlite3` – см. раздел документации «Использование подключения как контекстного менеджера» (*https://docs.python.org/3/library/sqlite3.html#using-the-connection-as-a-context-manager*);
- безопасная работа с блокировками, условными переменными и семафорами – как описано в документации по модулю `threading` (*https://docs.python.org/3/library/threading.html#using-locks-conditions-and-semaphores-in-the-with-statement*);
- организация специального окружения для арифметических операций с объектами `Decimal` – см. документацию по методу `decimal.localcontext` (*https://docs.python.org/3/library/decimal.html#decimal.localcontext*);
- внесение временных изменений в объекты для тестирования – см. документацию по функции `unittest.mock.patch` (*https://docs.python.org/3/library/unittest.mock.html#patch*).

Интерфейс контекстного менеджера состоит из методов `__enter__` и `__exit__`. В начале блока `with` вызывается метод `__enter__` контекстного менеджера. Когда блок `with` завершается естественным или иным образом, Python вызывает метод `__exit__` контекстного менеджера.

Самый распространенный пример `with` – гарантированное закрытие объекта файла, показанное в примере 18.1.

Пример 18.1. Использование объекта файла в качестве контекстного менеджера

```
>>> with open('mirror.py') as fp: ❶
...     src = fp.read(60) ❷
...
>>> len(src)
60
>>> fp ❸
<_io.TextIOWrapper name='mirror.py' mode='r' encoding='UTF-8'>
>>> fp.closed, fp.encoding ❹
```

```
(True, 'UTF-8')
>>> fp.read(60) ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

❶ Имя `fp` связано с открытым файлом, потому что метод `__enter__` объекта-файла возвращает `self`.
❷ Прочитать 60 символов Unicode из `fp`.
❸ Переменная `fp` все еще доступна – блоки `with` не определяют новую область видимости, в отличие от функций.
❹ Мы можем прочитать атрибуты объекта `fp`.
❺ Но выполнить операцию ввода-вывода для `fp` по завершении блока `with` нельзя, т. к. уже был вызван метод `TextIOWrapper.__exit__` и файл закрыт.

Маркер ❶ в примере 18.1 отмечает тонкий, но важный момент: объект контекстного менеджера – это результат вычисления выражения после слова `with`, но значение, связанное с переменной в части `as`, – результат вызова метода `__enter__` объекта контекстного менеджера.

В этом примере функция `open()` возвращает экземпляр класса `TextIOWrapper`, а его метод `__enter__` возвращает `self`. Но в другом классе метод `__enter__` может возвращать какой-то другой объект, не обязательно сам контекстный менеджер.

Когда поток управления покидает блок `with` любым способом, вызывается метод `__exit__` контекстного менеджера, а не объекта, возвращенного методом `__enter__`.

Часть `as` в предложении `with` необязательна. В случае `open` она необходима, чтобы получить ссылку на файл, но некоторые контекстные менеджеры возвращают `None` за неимением чего-то полезного.

В примере 18.2 показана работа шутливого контекстного менеджера, единственный смысл которого – подчеркнуть различие между самим менеджером и объектом, который возвращает его метод `__enter__`.

Пример 18.2. Тест класса контекстного менеджера `LookingGlass`

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what: ❶
...     print('Alice, Kitty and Snowdrop') ❷
...     print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what ❸
'JABBERWOCKY'
>>> print('Back to normal.') ❹
Back to normal.
```

❶ Контекстный менеджер – экземпляр класса `LookingGlass`; Python вызывает метод `__enter__` контекстного менеджера и связывает результат с переменной `what`.
❷ Напечатать `str`, а затем значение переменной `what`. Любая печатаемая строка выводится задом наперед.

❸ Блок `with` завершился. Как видим, метод `__enter__` вернул значение `'JABBERWOCKY'`, сохраненное в переменной `what`.
❹ Печатаемые строки больше не инвертируются.

В примере 18.3 показана реализация класса `LookingGlass`.

Пример 18.3. mirror.py: класс контекстного менеджера `LookingGlass`

```
import sys

class LookingGlass:

    def __enter__(self): ❶
        self.original_write = sys.stdout.write ❷
        sys.stdout.write = self.reverse_write ❸
        return 'JABBERWOCKY' ❹

    def reverse_write(self, text): ❺
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback): ❻
        sys.stdout.write = self.original_write ❼
        if exc_type is ZeroDivisionError: ❽
            print('Please DO NOT divide by zero!')
            return True ❾
        ❿
```

❶ Python вызывает `__enter__` с одним лишь аргументом `self`.
❷ Текущий метод `sys.stdout.write` сохраняется в атрибуте экземпляра для последующего использования.
❸ Подменить метод `sys.stdout.write` своим собственным.
❹ Вернуть строку `'JABBERWOCKY'`, просто чтобы было что поместить в переменную `what`.
❺ Наш метод `sys.stdout.write` инвертирует переданный аргумент `text` и вызывает сохраненную реализацию.
❻ Python вызывает метод `__exit__` с аргументами `None`, `None`, `None`, если не было ошибок; если же имело место исключение, то в аргументах передаются данные об исключении, описанные ниже.
❼ Восстановить исходный метод `sys.stdout.write`.
❽ Если исключение было и его тип – `ZeroDivisionError`, то напечатать сообщение…
❾ … и вернуть `True`, уведомляя интерпретатор о том, что исключение обработано.
❿ Если метод `__exit__` возвращает `None` или что-то, похожее на `False`, то исключение, возникшее внутри блока `with`, распространяется дальше.

Реальные приложения, перехватывающие стандартный вывод, обычно хотят временно подменить `sys.stdout` похожим на файл объектом, а затем восстановить исходное состояние. Именно это делает контекстный менеджер `contextlib.redirect_stdout` (*https://docs.python.org/3/library/contextlib.html#contextlib.redirect_stdout*): просто передайте ему похожий на файл объект, который подменит `sys.stdout`.

Интерпретатор вызывает метод `__enter__` без аргументов – если не считать неявного аргумента `self`. А методу `__exit__` передаются следующие три аргумента:

`exc_type`

Класс исключения (например, `ZeroDivisionError`).

`exc_value`

Объект исключения. Иногда в атрибуте `exc_value.args` можно найти параметры, переданные конструктору исключения, например сообщение об ошибке.

`traceback`

Объект `traceback`[1].

Детальное представление о работе контекстного менеджера дает пример 18.4, где объект `LookingGlass` используется вне блока `with`, чтобы можно было вручную вызвать его методы `__enter__` и `__exit__`.

Пример 18.4. Исследование `LookingGlass` без блока `with`

```
>>> from mirror import LookingGlass
>>> manager = LookingGlass() ❶
>>> manager # doctest: +ELLIPSIS
<mirror.LookingGlass object at 0x...>
>>> monster = manager.__enter__() ❷
>>> monster == 'JABBERWOCKY' ❸
eurT
>>> monster
'YKCOWREBBAJ'
>>> manager # doctest: +ELLIPSIS
>... ta tcejbo ssalGgnikooL.rorrim<
>>> manager.__exit__(None, None, None) ❹
>>> monster
'JABBERWOCKY'
```

❶ Создать и проинспектировать объект `manager`.
❷ Вызвать метод `__enter__()` контекстного менеджера и сохранить результат в переменной `monster`.
❸ Переменная `monster` содержит строку `'JABBERWOCKY'`. Идентификатор `True` инвертирован, потому что весь вывод на `stdout` проходит через метод `write`, который мы подменили в `__enter__()`.
❹ Вызвать `manager.__exit__`, чтобы восстановить исходный `stdout.write`.

Скобочные контекстные менеджеры в Python 3.10

В версии Python 3.10 появился новый, более мощный синтаксический анализатор (*https://peps.python.org/pep-0617/*), открывающий возможность для конструкций, которые были невозможны при старом анализаторе типа LL(1) (*https://en.wikipedia.org/wiki/LL_parser*). Одно такое синтаксическое улучшение – скобочные контекстные менеджеры:

[1] Три аргумента метода `__exit__` – это в точности то, что мы получили бы, вызвав метод `sys.exc_info()` (*https://docs.python.org/3/library/sys.html#sys.exc_info*) в блоке `finally` предложения `try/finally`. И это понятно, если вспомнить, что предложение `with` призвано заменить `try/finally` в большинстве случаев, а вызывать `sys.exc_info()` часто было необходимо, чтобы решить, какая требуется очистка.

```
with (
    CtxManager1() as example1,
    CtxManager2() as example2,
    CtxManager3() as example3,
):
...
```

До версии 3.10 мы должны были записывать это в виде вложенных блоков with.

В стандартную библиотеку входит пакет `contextlib`, содержащий полезные функции, классы и декораторы для построения, комбинирования и использования контекстных менеджеров.

Утилиты contextlib

Прежде чем начинать писать собственные классы контекстных менеджеров, прочитайте раздел документации по `contextlib` «Утилиты для контекстов, вводимых блоками with» (*https://docs.python.org/3/library/contextlib.html*). Быть может, то, что вы собираетесь делать, уже кем-то сделано. А возможно, существует класс или какой-то вызываемый объект, который облегчит вам работу.

Помимо уже упоминавшегося после примера 18.3 контекстного менеджера `redirect_stdout`, в версию Python 3.5 был добавлен менеджер `redirect_stderr` – он делает то же самое, но для выхода, направленного в `stderr`.

Модуль `contextlib` также включает другие классы и функции.

`closing`

Функция для построения контекстных менеджеров из объектов, которые предоставляют метод `close()`, но не реализуют интерфейс `__enter__`/`__exit__`.

`suppress`

Контекстный менеджер для временного игнорирования заданных исключений.

`nullcontext`

Контекстный менеджер, который не делает ничего, но упрощает логику вокруг объектов, которые, возможно, не реализуют подходящий контекстный менеджер. Он служит заместителем в случае, когда условный код перед блоком `with` может предоставить или не предоставить контекстный менеджер для этого предложения `with`. Добавлен в версии Python 3.7.

В модуле `contextlib` имеются классы и декоратор, которые применимы более широко, чем декораторы, упомянутые выше.

`@contextmanager`

Декоратор, который позволяет построить контекстный менеджер из простой генераторной функции, вместо того чтобы создавать класс и реализовывать интерфейс. См. раздел «Использование @contextmanager».

`AbstractContextManager`

Этот ABC формализует интерфейс контекстного менеджера и позволяет немного упростить процесс их создания благодаря наследованию. Добавлен в Python 3.6.

`ContextDecorator`

: Базовый класс для определения контекстных менеджеров на основе классов, которые можно использовать также в качестве декораторов функций, так что вся функция будет работать внутри управляемого контекста.

`ExitStack`

: Контекстный менеджер, который позволяет составлять композицию из переменного числа контекстных менеджеров. По выходе из блока `with` объект `ExitStack` вызывает методы `__exit__` запомненных контекстных менеджеров в порядке LIFO (последним вошел, первым обслужен). Этот класс применяется, когда заранее неизвестно количество открываемых блоков `with`, например в случае, когда одновременно открываются все файлы из произвольного списка.

В Python 3.7 в модуль `contextlib` добавились `AbstractAsyncContextManager`, `@asynccontextmanager` и `AsyncExitStack`. Они похожи на эквивалентные средства без префикса `async`, но предназначены для работы совместно с новым предложением `async with`, которое рассматривается в главе 21.

Из всех этих утилит чаще всего, безусловно, используется декоратор `@contextmanager`, поэтому уделим ему особое внимание. Этот декоратор интересен еще и тем, что предложение `yield` применяется в нем для целей, не связанных с итерированием.

Использование @contextmanager

Декоратор `@contextmanager` – элегантный и практичный инструмент, объединяющий три разных средства Python: декоратор функции, генератор и предложение `with`.

Использование `@contextmanager` уменьшает объем стереотипного кода создания контекстного менеджера: вместо того чтобы писать целый класс с методами `__enter__`/`__exit__`, мы просто реализуем генератор с одним предложением `yield`, порождающим значение, которое должен вернуть метод `__enter__`.

Если генератор снабжен декоратором `@contextmanager`, то `yield` разбивает тело функции на две части: все, что находится до `yield`, исполняется в начале блока `with`, когда интерпретатор вызывает метод `__enter__`; а все, что находится после `yield`, выполняется при вызове метода `__exit__` в конце блока.

В примере 18.5 класс `LookingGlass` из примера 18.3 заменен генераторной функцией.

Пример 18.5. mirror_gen.py: реализация контекстного менеджера с помощью генератора

```
import contextlib
import sys

@contextlib.contextmanager ❶
def looking_glass():
    original_write = sys.stdout.write ❷

    def reverse_write(text): ❸
        original_write(text[::-1])
```

```
    sys.stdout.write = reverse_write ❹
    yield 'JABBERWOCKY' ❺
    sys.stdout.write = original_write ❻
```

❶ Применить декоратор `contextmanager`.
❷ Сохранить исходный метод `sys.stdout.write`.
❸ Функция `reverse_write` сможет вызывать `original_write`, потому что та доступна в замыкании.
❹ Заменить `sys.stdout.write` функцией `reverse_write`.
❺ Отдать значение, которое будет связано с переменной в части `as` предложения `with`. В этой точке генератор приостанавливается на время выполнения блока `with`.
❻ Когда управление покидает блок `with` любым способом, выполнение функции возобновляется с места, следующего за `yield`; в данном случае мы восстанавливаем исходный метод `sys.stdout.write`.

В примере 18.6 показана функция `looking_glass` в действии.

Пример 18.6. Тест функции контекстного менеджера `looking_glass`

```
>>> from mirror_gen import looking_glass
>>> with looking_glass() as what: ❶
...     print('Alice, Kitty and Snowdrop')
...     print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what
'JABBERWOCKY'
>>> print('back to normal')
back to normal
```

❶ Единственное отличие от примера 18.2 – имя контекстного менеджера: `looking_glass` вместо `LookingGlass`.

Декоратор `contextlib.contextmanager` обертывает функцию классом, который реализует методы `__enter__` и `__exit__`[1].

Метод `__enter__` этого класса выполняет следующие действия:

1. Вызывает генераторную функцию, чтобы получить объект-генератор – назовем его `gen`.
2. Вызывает `next(gen)`, чтобы заставить генератор выполнить код до предложения `yield`.
3. Возвращает значение, отданное `next(gen)`, чтобы его можно было связать с переменной в части `as` блока `with`.

По завершении блока `with` метод `__exit__` выполняет следующие действия:

[1] Этот класс на самом деле называется `_GeneratorContextManager`. Если хотите узнать, как он работает, загляните в его исходный код (*https://github.com/python/cpython/blob/8afab2ebbc1b343cd88d058914cf622fe687a2be/ Lib/contextlib.py#L123*) в файле *Lib/contextlib.py* из дистрибутива Python 3.10.

1. Смотрит, было ли передано исключение в параметре `exc_type`; если да, вызывает `gen.throw(exception)`, в результате чего строка в теле генераторной функции, содержащая `yield`, возбуждает исключение.
2. В противном случае вызывает `next(gen)`, что приводит к выполнению части генераторной функции после `yield`.

В примере 18.5 есть дефект: если в теле блока `with` возникает исключение, то интерпретатор Python перехватывает его и повторно возбуждает в выражении `yield` внутри `looking_glass`. Но здесь нет никакой обработки исключений, поэтому функция `looking_glass` аварийно завершится, не восстановив исходный метод `sys.stdout.write` и оставив тем самым систему в некорректном состоянии.

В примере 18.7 добавлена специальная обработка исключения `ZeroDivisionError`, в результате чего код стал эквивалентен примеру 18.3, основанному на классе.

Пример 18.7. mirror_gen_exc.py: контекстный менеджер на основе генератора, реализующий обработку исключения, – внешнее поведение такое же, как в примере 18.3

```
import contextlib
import sys

@contextlib.contextmanager
def looking_glass():
    original_write = sys.stdout.write

    def reverse_write(text):
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    msg = '' ❶
    try:
        yield 'JABBERWOCKY'
    except ZeroDivisionError: ❷
        msg = 'Please DO NOT divide by zero!'
    finally:
        sys.stdout.write = original_write ❸
        if msg:
            print(msg) ❹
```

❶ Создать переменную для хранения возможного сообщения об ошибке; это первое изменение по сравнению с примером 18.5.
❷ Обработать исключение `ZeroDivisionError` – установить текст сообщения об ошибке.
❸ Восстановить исходный метод `sys.stdout.write`.
❹ Отобразить сообщение об ошибке, если оно не пусто.

Напомним, что, возвращая `True`, метод `__exit__` уведомляет интерпретатор о том, что он обработал исключение; в этом случае интерпретатор подавляет исключение. С другой стороны, если `__exit__` не вернул никакого значения явно, то интерпретатор получает значение по умолчанию `None` и распространяет исключение дальше. При наличии декоратора `@contextmanager` поведение по умолчанию изменяется на противоположное: метод `__exit__`, предоставляемый декоратором, предполагает, что любое исключение, посланное генератору, уже обработано и должно быть подавлено.

Наличие блока `try/finally` (или блока `with`) вокруг `yield` – неизбежная плата за использование `@contextmanager`, потому что невозможно заранее знать, что пользователи контекстного менеджера будут делать внутри блока `with`[1].

У декоратора `@contextmanager` есть одно малоизвестное свойство: снабженные им генераторы сами могут использоваться как декораторы[2]. Это возможно, потому что `@contextmanager` реализован с помощью класса `contextlib.ContextDecorator`.

В примере 18.8 показан контекстный менеджер `looking_glass` из примера 18.5, используемый как декоратор.

Пример 18.8. Контекстный менеджер `looking_glass` работает и как декоратор тоже

```
>>> @looking_glass()
... def verse():
...     print('The time has come')
...
>>> verse() ❶
emoc sah emit ehT
>>> print('back to normal') ❷
back to normal
```

❶ `looking_glass` работает до и после тела `verse`.
❷ Это подтверждает, что оригинальная функция `sys.write` была восстановлена.

Сравните пример 18.8 с примером 18.6, в котором `looking_glass` используется как контекстный менеджер.

Интересный практический пример использования `@contextmanager` за пределами стандартной библиотеки дает контекстный менеджер для перезаписи файла на месте, созданный Мартином Питерсом (*https://www.zopatista.com/python/2013/11/26/inplace-file-rewriting/*). В примере 18.9 показано, как он используется.

Пример 18.9. Контекстный менеджер для перезаписи файла на месте

```
import csv

with inplace(csvfilename, 'r', newline='') as (infh, outfh):
    reader = csv.reader(infh)
    writer = csv.writer(outfh)

    for row in reader:
        row += ['new', 'columns']
        writer.writerow(row)
```

Функция `inplace` – это контекстный менеджер, который предоставляет два описателя – `infh` и `outfh` – одного и того же файла, позволяющие одновременно читать и записывать файл. Это проще, чем функция `fileinput.input` из стандартной библиотеки (*https://docs.python.org/3/library/fileinput.html#fileinput.input*) (которая, кстати, тоже является контекстным менеджером).

---

[1] Это прямая цитата из замечания Леонардо Рохаэля, одного из технических рецензентов книги. Отлично сказано, Лео!

[2] По крайней мере, ни я, ни другие технические рецензенты не знали об этом, пока Калеб Хэттинг нас не просветил. Спасибо, Калеб!

Если вы хотите разобраться в исходном коде функции `inplace` (приведенном в вышеупомянутой статье), то ищите ключевое слово `yield`: все, что находится до него, связано с подготовкой контекста, т. е. созданием резервной копии и последующим открытием и отдачей описателей для чтения и записи, которые будут возвращены при вызове метода `__enter__`. В ходе обработки метода `__exit__` после слова `yield` закрываются описатели файлов, а если что-то пошло не так, то файл восстанавливается из резервной копии.

На этом завершается наш разговор о предложении `with` и контекстных менеджерах. Обратимся теперь к развернутому примеру, иллюстрирующему предложение `match/case`.

Сопоставление с образцом в lis.py: развернутый пример

В разделе «Сопоставление с последовательностями-образцами в интерпретаторе» главы 2 мы видели примеры последовательностей-образцов, взятых из функции `evaluate` интерпретатора *lis.py*, написанного Петером Норвигом и перенесенного на Python 3.10. В этом разделе я хочу представить более широкую картину устройства *lis.py* и заодно изучить все ветви `case` в `evaluate`, объяснив не только структуру образцов, но и что делает интерпретатор в каждой ветви.

Помимо желания более подробно поговорить об образцах, у меня было еще три причины для написания этого раздела.

1. Скрипт *lis.py* Норвига – прекрасный пример идиоматичного кода на Python.
2. Простота Scheme – отличный образчик проектирования языков.
3. Изучение работы интерпретатора позволило мне глубже понять Python и языки программирования вообще – как интерпретируемые, так и компилируемые.

Прежде чем переходить к Python-коду, попробуем Scheme на вкус, чтобы стал понятен смысл примера – это для тех, кто раньше не сталкивался с языками Scheme или Lisp.

Синтаксис Scheme

В Scheme, в отличие от Python, выражение и предложения не различаются. Все выражения записываются в префиксной нотации – `(+ x 13)` вместо `x + 13`. Она же используется для записи вызовов функций, например `(gcd x 13)`, и специальных форм, например `(define x 13)`, которые в Python были бы записаны в виде выражения присваивания `x = 13`. Нотация, применяемая в Scheme и большинстве диалектов Lisp, называется *S-выражениями*[1].

В примере 18.10 приведен пример простой программы на Scheme.

[1] Часто люди недовольны изобилием скобок в Lisp, но при надлежащих отступах и наличии хорошего редактора эта проблема теряет остроту. Основной проблемой в плане удобочитаемости является использование одной и той же нотации `(f ...)` для вызовов функций и специальных форм вида `(define ...)`, `(if ...)` и `(quote ...)`, которые ведут себя совсем не так, как функции.

Пример 18.10. Вычисление наибольшего общего делителя на Scheme

```
(define (mod m n)
    (- m (* n (quotient m n))))

(define (gcd m n)
    (if (= n 0)
        m
        (gcd n (mod m n))))

(display (gcd 18 45))
```

В этом примере мы видим три выражения Scheme: два определения функций – `mod` и `gcd` – и вызов функции `display`, которая выводит 9, результат выражения `(gcd 18 45)`. В примере 18.11 показан эквивалентный код на Python (он короче, чем объяснение *алгоритма Евклида* (*https://en.wikipedia.org/wiki/Euclidean_algorithm*) на естественном языке).

Пример 18.11. То же, что пример 18.10, но на Python

```
def mod(m, n):
    return m - (m // n * n)

def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, mod(m, n))

print(gcd(18, 45))
```

Если бы я писал идиоматичный код, то воспользовался бы оператором `%`, а не изобретал бы заново `mod`, и вместо рекурсии было бы эффективнее использовать цикл `while`. Но я хотел показать для сравнения оба определения функции, сделав код максимально похожим, чтобы вам было проще читать код на Scheme.

В Scheme нет команд итеративного управления потоком типа `while` или `for`. Итерирование всегда производится с помощью рекурсии. Обратите внимание, что в примерах на Scheme и Python нет присваиваний. Повсеместное использование рекурсии и минимум присваиваний – отличительные признаки программирования в функциональном стиле[1].

Теперь перейдем к рассмотрению кода *lis.py* на Python 3.10. Полный исходный код имеется в каталоге *18-with-match/lispy/py3.10/* репозитория на GitHub по адресу *fluentpython/example-code-2e* (*https://github.com/fluentpython/example-code-2e/tree/ master/18-with-match/lispy/py3.10/*).

Предложения импорта и типы

В примере 18.12 показаны первые строки скрипта *lis.py*. Для использования `TypeAlias` и оператора объединения типов `|` необходима версия Python 3.10.

[1] Чтобы замена итерации рекурсией была практична и эффективна, в Scheme и других функциональных языках реализована корректная обработка *хвостовых вызовов*. Подробнее об этом написано в разделе «Поговорим».

Пример 18.12. lis.py: начало файла

```
import math
import operator as op
from collections import ChainMap
from itertools import chain
from typing import Any, TypeAlias, NoReturn

Symbol: TypeAlias = str
Atom: TypeAlias = float | int | Symbol
Expression: TypeAlias = Atom | list
```

Типы определены следующим образом.

`Symbol`

Просто псевдоним `str`. В *lis.py* `Symbol` используется для идентификаторов; не существует строкового типа данных с такими операциями, как срез, расщепление и т. д.[1]

`Atom`

Простой синтаксический элемент, например число или `Symbol`, – в противоположность структуре, состоящей из нескольких частей, как, например, список.

`Expression`

Строительными блоками программ на Scheme являются выражения, состоящие из атомов и списков, возможно, вложенных.

Синтаксический анализатор

Анализатор Норвига состоит из 36 строк кода, демонстрирующего возможности Python применительно к обработке простого рекурсивного синтаксиса S-выражений – без строковых данных, комментариев, макросов и других средств стандартного Scheme, усложняющих синтаксический разбор (пример 18.13).

Пример 18.13. lis.py: основные функции синтаксического разбора

```
def parse(program: str) -> Expression:
    "Читать выражение Scheme из строки."
    return read_from_tokens(tokenize(program))

def tokenize(s: str) -> list[str]:
    "Преобразовать строку в список лексем."
    return s.replace('(', ' ( ').replace(')', ' ) ').split()

def read_from_tokens(tokens: list[str]) -> Expression:
    "Читать выражение из последовательности лексем."
    # дополнительный код синтаксического разбора опущен
```

[1] Но во втором интерпретаторе Норвига, *lispy.py* (*https://github.com/fluentpython/example-code-2e/blob/master/18-with-match/lispy/original/lispy.py*), поддерживаются строки как тип данных, а также такие продвинутые средства, как синтаксические макросы, продолжения и хвостовые вызовы. Однако *lispy.py* почти в три раза длиннее *lis.py* и разобраться в нем гораздо труднее.

Главная функция в этой группе – `parse`, она принимает S-выражение в виде `str` и возвращает объект типа `Expression`, определенного в примере 18.12: `Atom` или список `list`, который может содержать атомы и вложенные списки.

В функции `tokenize` Норвиг применил ловкий трюк: он добавляет пробелы до и после каждой скобки во входных данных, а затем расщепляет строку, получая список синтаксических лексем, в котором `'('` и `')'` представлены отдельными лексемами. Этот прием работает, потому что в сокращенном Scheme нет типа строки, так что каждая скобка `'('` или `')'` является ограничителем выражения. Код рекурсивного разбора находится в 14-строчной функции `read_from_tokens`, его можно найти в каталоге *fluentpython/example-code-2e* в репозитории. Я его опущу, потому что хочу уделить больше внимания другим частям интерпретатора.

Ниже приведены некоторые тесты, взятые из файла *lispy/py3.10/examples_test.py*:

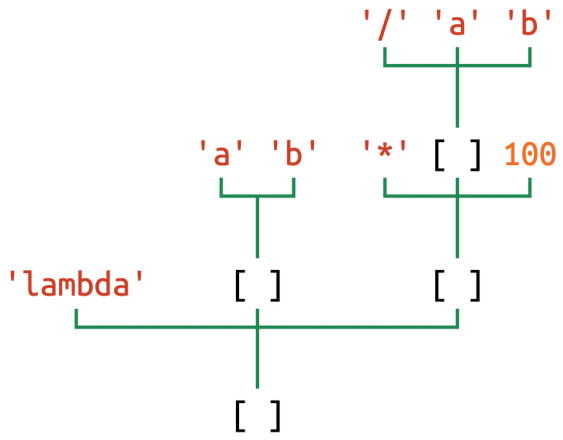
```
>>> from lis import parse
>>> parse('1.5')
1.5
>>> parse('ni!')
'ni!'
>>> parse('(gcd 18 45)')
['gcd', 18, 45]
>>> parse('''
... (define double
... (lambda (n)
... (* n 2)))
... ''')
['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]
```

Правила разбора для этого подмножества Scheme просты:

1. Лексема, похожая на число, разбирается как `float` или `int`.
2. Всё, кроме `'('` и `')'`, разбирается как `Symbol` – строка `str`, используемая в роли идентификатора. Сюда входят, в частности, `+`, `set!` и `make-counter`, все это допустимые идентификаторы в Scheme, но не в Python.
3. Выражения внутри `'('` и `')'` разбираются рекурсивно как списки, содержащие атомы, или как вложенные списки, которые могут содержать атомы и дополнительные вложенные списки.

В терминологии интерпретатора Python результатом `parse` является AST (абстрактное синтаксическое дерево): удобное представление программы на Scheme в виде вложенных списков, образующих древовидную структуру, в которой самый внешний список является стволом, внутренние списки – ветвями, а атомы – листьями (рис. 18.1).

```
(lambda (a b) (* (/ a b) 100))
```




```
['lambda', ['a', 'b'], ['*', ['/', 'a', 'b'], 100]]
```

Рис. 18.1. Лямбда-выражение, представленное в виде исходного кода (конкретный синтаксис), в виде дерева и в виде последовательности объектов Python (абстрактный синтаксис)

Класс Environment

Класс `Environment` расширяет `collections.ChainMap`, добавляя метод `change` для обновления значения в одном из связанных словарей, которые в экземплярах `ChainMap` хранятся в списке отображений: атрибуте `self.maps`. Метод `change` необходим для поддержки формы Scheme (`set!` ...), описанной ниже; см. пример 18.14.

Пример 18.14. lis.py: класс `Environment`

```
class Environment(ChainMap[Symbol, Any]):
    "ChainMap, позволяющий обновлять элемент на месте."

    def change(self, key: Symbol, value: Any) -> None:
        "Найти, где определен ключ, и изменить там значение."
        for map in self.maps:
            if key in map:
                map[key] = value # type: ignore[index]
                return
        raise KeyError(key)
```

Заметим, что метод `change` обновляет только уже существующие ключи[1]. Попытка изменить ненайденный ключ приводит к исключению `KeyError`.

[1] Комментарий `# type: ignore[index]` поставлен из-за проблемы *typeshed* #6042 (*https://github.com/python/typeshed/issues/6042*), которая на момент рецензирования этой главы оставалась неразрешенной. Класс `ChainMap` аннотирован как `MutableMapping`, но аннотация типа в атрибуте `maps` говорит, что это список объектов `Mapping`, что косвенно делает весь `ChainMap` неизменяемым с точки зрения Муру.

Следующий тест показывает, как работает `Environment`:

```
>>> from lis import Environment
>>> inner_env = {'a': 2}
>>> outer_env = {'a': 0, 'b': 1}
>>> env = Environment(inner_env, outer_env)
>>> env['a'] ❶
2
>>> env['a'] = 111 ❷
>>> env['c'] = 222
>>> env
Environment({'a': 111, 'c': 222}, {'a': 0, 'b': 1})
>>> env.change('b', 333) ❸
>>> env
Environment({'a': 111, 'c': 222}, {'a': 0, 'b': 333})
```

❶ При чтении значений `Environment` работает как `ChainMap`: ключи ищутся во вложенных отображениях слева направо. Именно поэтому значение `a` в `outer_env` маскируется значением в `inner_env`.

❷ Присваивание с применением `[]` перезаписывает существующие или вставляет новые элементы, но всегда в первом отображении, в данном случае – `inner_env`.

❸ `env.change('b', 333)` ищет ключ `'b'` и присваивает ему новое значение на месте, в `outer_env`.

Далее показана функция `standard_env()`, которая строит и возвращает экземпляр `Environment` с уже загруженными предопределенными функциями. Это напоминает модуль `__builtins__` в Python, который всегда доступен (пример 18.15).

Пример 18.15. lis.py: `standard_env()` строит и возвращает глобальное окружение

```
def standard_env() -> Environment:
    "Окружение, содержащее некоторые стандартные процедуры Scheme."
    env = Environment()
    env.update(vars(math)) # sin, cos, sqrt, pi, ...
    env.update({
        '+': op.add,
        '-': op.sub,
        '*': op.mul,
        '/': op.truediv,
        # определения других операторов опущены
        'abs': abs,
        'append': lambda *args: list(chain(*args)),
        'apply': lambda proc, args: proc(*args),
        'begin': lambda *x: x[-1],
        'car': lambda x: x[0],
        'cdr': lambda x: x[1:],
        # определения других функций опущены
        'number?': lambda x: isinstance(x, (int, float)),
        'procedure?': callable,
        'round': round,
        'symbol?': lambda x: isinstance(x, Symbol),
    })
    return env
```

Таким образом, в отображение `env` загружаются:

- все функции из модуля Python `math`;
- избранные операторы из модуля Python `op`;
- простые, но мощные функции, построенные с помощью лямбда-выражений Python (`lambda`);
- переименованные встроенные функции Python (например, `callable` называется `procedure?`), а также функции с такими же именами, как в Python (например, `round`).

Цикл REPL

Цикл REPL (read-eval-print-loop – цикл чтения, вычисления, печати) в коде Норвига легко понять, но назвать его дружелюбным к пользователю сложно (см. пример 18.16). Если скрипту *lis.py* не передано никаких аргументов, то `main()`, определенная в конце модуля, вызывает функцию `repl()`. В ответ на приглашение `lis.py>` мы должны ввести правильное и полное выражение; если не закрыть одну скобку, то *lis.py* аварийно завершается[1].

Пример 18.16. Функции цикла REPL

```
def repl(prompt: str = 'lis.py> ') -> NoReturn:
    "Цикл приглашение-чтение-вычисление-печать."
    global_env = Environment({}, standard_env())
    while True:
        ast = parse(input(prompt))
        val = evaluate(ast, global_env)
        if val is not None:
            print(lispstr(val))

def lispstr(exp: object) -> str:
    "Преобразовать объект Python назад в строку, понятную Lisp."
    if isinstance(exp, list):
        return '(' + ' '.join(map(lispstr, exp)) + ')'
    else:
        return str(exp)
```

Приведем краткое пояснение к этим двум функциям:

`repl(prompt: str = 'lis.py> ') -> NoReturn`

Вызывает `standard_env()`, чтобы сделать доступными функции из глобального окружения, затем входит в бесконечный цикл, где читает и разбирает входные строки, вычисляет их в глобальном контексте и отображает результат, если он равен `None`. Функция `evaluate` может модифицировать `global_env`. Например, если пользователь определяет новую глобальную переменную или именованную функцию, они сохраняются в первом отображении окружения – пустом словаре `dict`, переданном конструктору `Environment` в первой строчке `repl`.

[1] Изучая скрипты Норвига *lis.py* и *lispy.py*, я разветвил проект и в свою версию *mylis* кое-что добавил, например мой цикл REPL принимает частичные S-выражения и выводит приглашение для ввода продолжения. Точно так же в Python цикл REPL понимает, что выражение не закончено, и предлагает вспомогательное приглашение (`...`) до тех пор, пока не получит полное выражение или предложение, которое можно вычислить. Кроме того, *mylis* аккуратно обрабатывает некоторые ошибки, но все равно «уронить» его легко. Он далеко не так надежен, как цикл REPL в Python.

`lispstr(exp: object) -> str`

Функция, обратная `parse`: получив объект Python, представляющий выражение, `parse` возвращает его исходный код на Scheme. Например, для `['+', 2, 3]` будет возвращено `'(+ 2 3)'`.

Вычислитель

Теперь мы можем оценить красоту вычислителя выражений в коде Норвига, который сделан немного симпатичнее благодаря использованию `match/case`. Функция `evaluate`, показанная в примере 18.17, принимает объект `Expression`, построенный `parse`, и объект типа `Environment`.

Тело `evaluate` состоит из одного предложения `match`, в котором субъектом является выражение `exp`. В образцах `case` с удивительной ясностью выражены синтаксис и семантика Scheme.

Пример 18.17. `evaluate` принимает выражение и вычисляет его значение

```
KEYWORDS = ['quote', 'if', 'lambda', 'define', 'set!']

def evaluate(exp: Expression, env: Environment) -> Any:
    "Вычислить выражение в заданном окружении."
    match exp:
        case int(x) | float(x):
            return x
        case Symbol(var):
            return env[var]
        case ['quote', x]:
            return x
        case ['if', test, consequence, alternative]:
            if evaluate(test, env):
                return evaluate(consequence, env)
            else:
                return evaluate(alternative, env)
        case ['lambda', [*parms], *body] if body:
            return Procedure(parms, body, env)
        case ['define', Symbol(name), value_exp]:
            env[name] = evaluate(value_exp, env)
        case ['define', [Symbol(name), *parms], *body] if body:
            env[name] = Procedure(parms, body, env)
        case ['set!', Symbol(name), value_exp]:
            env.change(name, evaluate(value_exp, env))
        case [func_exp, *args] if func_exp not in KEYWORDS:
            proc = evaluate(func_exp, env)
            values = [evaluate(arg, env) for arg in args]
            return proc(*values)
        case _:
            raise SyntaxError(lispstr(exp))
```

Рассмотрим, что делает каждая ветвь `case`. В некоторых случаях я добавил комментарии, показывающие, как S-выражение будет соответствовать образцу при построении списка Python. Тесты, взятые из файла *examples_test.py* (*https://github.com/fluentpython/example-code-2e/blob/00e4741926e1b771ee7c753148b1415c0bd12e39/02-array-seq/lispy/py3.10/examples_test.py*), демонстрируют каждую ветвь.

Вычисление чисел

```
case int(x) | float(x):
    return x
```

Субъект:

Экземпляр `int` или `float`.

Действие:

Вернуть прочитанное значение.

Пример:

```
>>> from lis import parse, evaluate, standard_env
>>> evaluate(parse('1.5'), {})
1.5
```

Вычисление символов

```
case Symbol(var):
    return env[var]
```

Субъект:

Экземпляр `Symbol`, т. е. строка, используемая как идентификатор.

Действие:

Найти `var` в `env` и вернуть значение.

Примеры:

```
>>> evaluate(parse('+'), standard_env())
<встроенная функция add>
>>> evaluate(parse('ni!'), standard_env())
Traceback (most recent call last):
    ...
KeyError: 'ni!'
```

(quote ...)

Специальная форма `quote` позволяет рассматривать атомы и списки как данные, а не подлежащие вычислению выражения.

```
# (quote (99 bottles of beer))
case ['quote', x]:
    return x
```

Субъект:

Список, начинающийся символом `'quote'`, за которым следует одно выражение `x`.

Действие:

Вернуть `x` без вычисления.

Примеры:

```
>>> evaluate(parse('(quote no-such-name)'), standard_env())
'no-such-name'
>>> evaluate(parse('(quote (99 bottles of beer))'), standard_env())
[99, 'bottles', 'of', 'beer']
```

```
>>> evaluate(parse('(quote (/ 10 0))'), standard_env())
['/', 10, 0]
```

Не будь `quote`, каждое из этих выражений возбудило бы исключение:

- `no-such-name` было бы не найдено в окружении, что привело бы к ошибке `KeyError`;
- `(99 bottles of beer)` нельзя вычислить, потому что число 99 – не `Symbol`, содержащий имя специальной формы, оператора или функции;
- `(/ 10 0)` привело бы к исключению `ZeroDivisionError`.

Почему в языках есть зарезервированные слова

Форму `quote`, какой бы простой она ни была, нельзя реализовать в Scheme в виде функции. Ее специальная особенность заключается в том, чтобы помешать интерпретатору вычислять `(f 10)` в выражении `(quote (f 10))`: результатом является просто список, содержащий `Symbol` и `int`. С другой стороны, при вызове функции, например `(abs (f 10))`, интерпретатор сначала вычисляет `(f 10)`, а затем вызывает `abs`. Именно поэтому `quote` является зарезервированным словом: оно должно обрабатываться как специальная форма.

В общем случае зарезервированные слова нужны:

- чтобы реализовать специальные правила, как в случае `quote` и `lambda`, – ни то, ни другое не вычисляет своих подвыражений;
- чтобы изменить поток управления, как в случае `if` и вызовов функций, – это тоже специальные правила вычисления;
- для управления окружением, как в случае `define` и `set`.

По тем же причинам в Python и в других языках программирования необходимы зарезервированные слова. Подумайте о том, для чего Python нужны слова `def`, `if`, `yield`, `import` и `del`.

(if ...)

```
        # (if (< x 0) 0 x)
        case ['if', test, consequence, alternative]:
            if evaluate(test, env):
                return evaluate(consequence, env)
            else:
                return evaluate(alternative, env)
```

Субъект:

Список, в котором первым элементом является `'if'`, а за ним следуют три выражения: `test`, `consequence` и `alternative`.

Действие:

Вычислить `test`:

- если `true`, то вычислить `consequence` и вернуть его значение;
- в противном случае вычислить `alternative` и вернуть его значение.

Примеры:

```
>>> evaluate(parse('(if (= 3 3) 1 0))'), standard_env())
1
>>> evaluate(parse('(if (= 3 4) 1 0))'), standard_env())
0
```

Ветви `consequence` и `alternative` должны содержать ровно одно выражение. Если необходимо несколько выражений, то их можно объединить: `(begin exp1 exp2…)`; в *lis.py* `begin` является функцией. См. пример 18.15.

(lambda ...)

В Scheme форма `lambda` определяет анонимные функции. У нее нет ограничений, присущих лямбда-выражениям в Python: любую функцию, которую вообще можно записать в Scheme, можно записать и с применением синтаксиса `(lambda …)`.

```
# (lambda (a b) (// (+ a b) 2))
case ['lambda' [*parms], *body] if body:
    return Procedure(parms, body, env)
```

Субъект:

Список, начинающийся со слова `'lambda'`, за которым следуют:

- список, содержащий нуль или более имен параметров;
- одно или несколько выражений, собранных в `body` (охранное условие гарантирует, что `body` не пусто).

Действие:

Создать и вернуть новый экземпляр класса `Procedure`, содержащий имена параметров, список выражений в качестве тела и текущее окружение.

Пример:

```
>>> expr = '(lambda (a b) (* (/ a b) 100))'
>>> f = evaluate(parse(expr), standard_env())
>>> f # doctest: +ELLIPSIS
<lis.Procedure object at 0x...>
>>> f(15, 20)
75.0
```

Класс `Procedure` реализует концепцию замыкания: вызываемый объект, в котором хранятся имена параметров, тело функции и ссылка на окружение, в котором функция определена. Код `Procedure` мы рассмотрим чуть позже.

(define ...)

У ключевого слова `define` есть две разные синтаксические формы. Самая простая такова:

```
# (define half (/ 1 2))
case ['define', Symbol(name), value_exp]:
    env[name] = evaluate(value_exp, env)
```

Субъект:

Строка начинается словом `'define'`, затем следуют `Symbol` и выражение.

Действие:

Вычислить выражение и поместить его значение в `env`, используя `name` как ключ.

Пример:

```
>>> global_env = standard_env()
>>> evaluate(parse('(define answer (* 7 6))'), global_env)
>>> global_env['answer']
42
```

Тест для этой ветви `case` создает `global_env`, чтобы можно было проверить, что `evaluate` помещает `answer` в окружение.

Эту простую форму `define` можно использовать для создания переменных или для связывания имен с анонимными функциями, когда в роли `value_exp` выступает `(lambda …)`.

В стандартном `Scheme` имеется сокращенная форма для определения именованных функций, а именно:

```
# (define (average a b) (/ (+ a b) 2))
case ['define', [Symbol(name), *parms], *body] if body:
    env[name] = Procedure(parms, body, env)
```

Субъект:

Список начинается с `'define'`, за которым следуют:

- список, начинающийся с `Symbol(name)`, за которым следует нуль или более элементов, собранных в список с именем `parms`;
- одно или более выражений, собранных в список `body` (охранное условие гарантирует, что `body` не пуст).

Действие:

- Создать новый экземпляр `Procedure` с заданными именами параметров, списком выражений в качестве тела и текущим окружением;
- поместить `Procedure` в `env`, используя `name` как ключ.

Тест в примере 18.18 определяет функцию с именем `%`, которая вычисляет процент и помещает его в `global_env`.

Пример 18.18. Определение функции `%`, которая вычисляет процент

```
>>> global_env = standard_env()
>>> percent = '(define (% a b) (* (/ a b) 100))'
>>> evaluate(parse(percent), global_env)
>>> global_env['%'] # doctest: +ELLIPSIS
<lis.Procedure object at 0x...>
>>> global_env['%'](170, 200)
85.0
```

После вызова `evaluate` мы проверяем, что имя `%` связано с `Procedure`, которая принимает два числовых аргумента и возвращает процент.

Образец для второй ветви `define` не гарантирует, что все элементы `parms` являются экземплярами `Symbol`. Я должен был бы сам проверить это перед построением `Procedure`, но не стал – чтобы следить за кодом было так же просто, как за кодом Норвига.

(set! ...)

Форма `set!` изменяет значение ранее определенной переменной[1].

```
# (set! n (+ n 1))
case ['set!', Symbol(name), value_exp]:
    env.change(name, evaluate(value_exp, env))
```

Субъект:

Список, начинающийся с `'set!'`, за которым следует `Symbol` и выражение.

Действие:

Заменить значение `name` в `env` результатом вычисления выражения.

Метод `Environment.change` обходит сцепленные окружения в порядке от локальных к глобальному и заменяет первое вхождение `name` новым значением. Если бы мы не стали реализовывать ключевое слово `'set!'`, то могли бы использовать класс Python `ChainMap` в качестве типа `Environment` всюду в этом интерпретаторе.

Ключевые слова nonlocal в Python и set! в Scheme решают одну и ту же проблему

Использование формы `set!` имеет прямое отношение к использованию ключевого слова `nonlocal` в Python: благодаря объявлению `nonlocal x` присваивание `x = 10` может обновить переменную `x`, ранее определенную вне локальной области видимости. Не будь объявления `nonlocal x`, это предложение всегда создавало бы локальную переменную, как мы видели в разделе «Объявление nonlocal Declaration» главы 9.

Аналогично форма `(set! x 10)` изменяет переменную `x`, которая, возможно, ранее была определена вне локального окружения функции. Напротив, переменная `x` в форме `(define x 10)` всегда локальна, она создается или изменяется в локальном окружении.

И `nonlocal`, и `(set! …)` необходимы, чтобы можно было изменять состояние программы, хранящееся в переменных в замыкании. В примере 9.13 демонстрировалось применение `nonlocal` для реализации функции, вычисляющей накопительное среднее, для чего число элементов `count` и частичная сумма `total` хранились в замыкании. Здесь та же идея выражена на подмножестве Scheme, реализованном в *lis.py*:

```
(define (make-averager)
    (define count 0)
    (define total 0)
    (lambda (new-value)
        (set! count (+ count 1))
        (set! total (+ total new-value))
```

[1] Присваивание – одна из первых вещей, которым учат во многих пособиях по программированию, но в лучшей из известных мне книг по Scheme, Abelson et al. «Structure and Interpretation of Computer Programs» (MIT Press), она же SCIP, или «Книга с мудрецом», описание `set!` появляется только на 220-й странице. Кодирование в функциональном стиле позволяет продвинуться очень далеко без изменения состояния, столь типичного для императивного и объектно-ориентированного программирования.

```
            (/ total count)
        )
    )
    (define avg (make-averager)) ❶
    (avg 10) ❷
    (avg 11) ❸
    (avg 15) ❹
```

❶ Создает новое замыкание с внутренней функцией, определенной с помощью `lambda`, и переменными `count` и `total`, инициализированными значением 0; связывает с замыканием имя `avg`.

❷ Возвращает 10.0.

❸ Возвращает 10.5.

❹ Возвращает 12.0.

Приведенный выше код – один из тестов в файле *lispy/py3.10/examples_test.py* (*https://github.com/fluentpython/example-code-2e/blob/master/18-with-match/lispy/py3.10/examples_test.py*).

Теперь перейдем к вызову функции.

Вызов функции

```
    # (gcd (* 2 105) 84)
    case [func_exp, *args] if func_exp not in KEYWORDS:
        proc = evaluate(func_exp, env)
        values = [evaluate(arg, env) for arg in args]
        return proc(*values)
```

Субъект:

Список, содержащий один или несколько элементов.
Охранное условие гарантирует, что `func_exp` не является ни одним из ключевых слов `['quote', 'if', 'define', 'lambda', 'set!']`, перечисленных до `evaluate` в примере 18.17.
Образец сопоставляется с любым списком, содержащим одно или более выражений, при этом первое выражение связывается с `func_exp`, а остальные со списком `args`, который может быть пустым.

Действие:

- Вычислить `func_exp` для получения функции `proc`;
- вычислить каждый элемент в `args` для построения списка значений аргументов;
- вызвать `proc`, передав значения в качестве отдельных аргументов, и вернуть результат.

Пример:

```
>>> evaluate(parse('(% (* 12 14) (- 500 100))'), global_env)
42.0
```

Этот тест – продолжение примера 18.18: предполагается, что в `global_env` имеется функция с именем `%`. Аргументы, передаваемые `%`, – арифметические выражения, чтобы подчеркнуть, что аргументы вычисляются до вызова функции.

Охранное условие в этой ветви `case` необходимо, потому что образец `[func_exp, *args]` сопоставляется с любой последовательностью-субъектом, содержащей один или более элементов. Однако если `func_exp` – ключевое слово, но субъект не сопоставился ни с одной предыдущей ветвью, то мы имеем синтаксическую ошибку.

Обработка синтаксических ошибок

Если субъект `exp` не сопоставился ни с одним из рассмотренных выше образцов в ветвях `case`, то в универсальной ветви возбуждается исключение `SyntaxError`:

```
        case _:
            raise SyntaxError(lispstr(exp))
```

Ниже приведен пример некорректной формы `(lambda …)`, которая вызывает ошибку `SyntaxError`:

```
>>> evaluate(parse('(lambda is not like this)'), standard_env())
Traceback (most recent call last):
  ...
SyntaxError: (lambda is not like this)
```

Если бы в ветви `case` для вызова функции не было охранного условия, отвергающего ключевые слова, то выражение `(lambda is not like this)` обрабатывалось бы как вызов функции, что привело бы к ошибке `KeyError`, потому что `'lambda'` не является частью окружения – точно так же, как `lambda` не является встроенной функцией в Python.

Procedure: класс, реализующий замыкание

Класс `Procedure` вполне можно было бы назвать `Closure`, потому что это именно то, для чего он предназначен: хранить определение функции вместе с окружением. Определение функции включает имена параметров и выражения, составляющие тело функции. Окружение используется при вызове функции, чтобы предоставить значения *свободных переменных*: переменных, которые встречаются в теле функции, но не являются ни параметрами, ни локальными переменными, ни глобальными переменными. Мы знакомились с концепцией *замыкания* и *свободной переменной* в разделе «Замыкания» главы 9.

Мы видели, как замыкания используются в Python, но теперь можем копнуть глубже и посмотреть, как замыкание реализовано в *lis.py*:

```
class Procedure:
    "Определенная пользователем процедура Scheme."

    def __init__( ❶
        self, parms: list[Symbol], body: list[Expression], env: Environment):
        self.parms = parms ❷
        self.body = body
        self.env = env

    def __call__(self, *args: Expression) -> Any: ❸
        local_env = dict(zip(self.parms, args)) ❹
        env = Environment(local_env, self.env) ❺
        for exp in self.body: ❻
            result = evaluate(exp, env)
        return result ❼
```

❶ Вызывается, когда функция определена с помощью форм `lambda` или `define`.
❷ Сохранить имена параметров, выражения в теле и окружение для последующего использования.
❸ Вызывается при обращении к `proc(*values)` в последней строке кода ветви `case [func_exp, *args]`.
❹ Построить отображение `local_env`, используя `self.parms` как имена локальных переменных, а `args` как значения.
❺ Построить новое объединенное окружение `env`, поместив в начало `local_env`, а за ним `self.env` – окружение, которое было сохранено в момент определения функции.
❻ Обойти все выражения в `self.body`, вычисляя их в контексте объединенного окружения `env`.
❼ Вернуть результат последнего вычисленного выражения.

В *lis.py* есть еще две простые функции после `evaluate`: `run` читает полную программу Scheme и выполняет ее, а `main` вызывает `run` или `repl` в зависимости от параметров в командной строке – так же поступает Python. Я не буду описывать эти функции, потому что в них нет ничего нового. Моя цель – просто разделить с вами восхищение красотой небольшого интерпретатора Норвига, подробнее пояснить, как работают замыкания, и убедить в том, что `match/case` стало прекрасным добавлением в Python.

И в заключение этого пространного раздела о сопоставлении с образцами формализуем понятие OR-образца.

Использование OR-образцов

Последовательность образцов, разделенных знаком `|`, называется *OR-образцом* (*https://peps.python.org/pep-0634/#or-patterns*): сопоставление с ним завершается успешно, если удалось сопоставить субъект хотя бы с одним из подобразцов. Образец в разделе «Вычисление чисел» выше является OR-образцом:

```
case int(x) | float(x):
    return x
```

Во всех подобразцах OR-образца должны использоваться одни и те же переменные. Это ограничение необходимо, чтобы гарантировать, что переменные доступны охранному выражению и телу ветви `case` вне зависимости от того, с каким из подобразцов произошло сопоставление.

В контексте ветви `case` оператор `|` имеет специальную семантику. Он не активирует специальный метод `__or__`, как при обработке выражений вида `a | b` в других контекстах, где он перегружен для выполнения таких операций, как объединение множеств или поразрядное OR целых чисел – в зависимости от операндов.

OR-образец может встречаться не только на верхнем уровне образца. Его можно использовать также в подобразцах `|`. Например, если мы захотим, что-

бы *lis.py* принимал греческую букву λ (лямбда)[1] наряду с ключевым словом `lambda`, то сможем переписать этот образец в таком виде:

```
# (λ (a b) (* (+ a b) 2) )
case ['lambda' | 'λ', [*parms], *body] if body:
    return Procedure(parms, body, env)
```

Теперь мы наконец-то можем перейти к третьей и последней теме этой главы: необычным местам, в которых может встречаться фраза `else` в Python.

Делай то, потом это: блоки else вне if

Это не секрет, а недооцененное средство языка: часть `else` может встречаться не только в предложениях `if`, но также в `for`, `while` и `try`.

Семантика `for/else`, `while/else` и `try/else` похожа, но резко отличается от семантики `if/else`. Поначалу слово `else` мешало мне по-настоящему понять смысл этих средств, но в конце концов я их освоил.

Правила таковы:

`for`

Блок `else` выполняется, только если цикл `for` дошел до конца (т. е. не было преждевременного выхода с помощью `break`).

`while`

Блок `else` выполняется, только если цикл `while` завершился вследствие того, что условие приняло ложное значение (а не в результате выхода с помощью `break`).

`try`

Блок `else` выполняется, только если в блоке `try` не возникало исключение. В официальной документации (*https://docs.python.org/3/reference/ compound_stmts.html*) также сказано: «Исключения, возникшие в части `else`, не обрабатываются в предшествующих частях `except`».

В любом случае часть `else` не выполняется и тогда, когда исключение либо одно из предложений, `return`, `break` или `continue`, приводят к передаче управления вовне главного блока составного предложения.

Я считаю, что выбор ключевого слова `else` крайне неудачен во всех случаях, кроме `if`. Оно подразумевает взаимно исключающие альтернативы, например: «Выполни этот цикл, иначе сделай то-то», однако семантика `else` в циклах прямо противоположна: «Выполни этот цикл, а затем сделай то-то». Таким образом, более подходящим словом было бы `then` – оно, кстати, имеет смысл и в контексте `try`: «Попробуй это, а затем сделай то». Однако добавление нового ключевого слова означало бы несовместимое изменение языка – непростое решение.

[1] Официальное название символа Unicode λ (U+03BB) ГРЕЧЕСКАЯ СТРОЧНАЯ БУКВА ЛЯМДА (GREEK SMALL LETTER LAMDA). Это не опечатка: в базе данных Unicode символ называется именно «лямда» без «б». Согласно статье в англоязычной Википедии «Lambda», консорциум Unicode принял это написание, поскольку таковы были «пожелания, высказанные национальным органом Греции».

Использование `else` в этих предложениях часто упрощает чтение кода и позволяет отказаться от установки всяких флагов и добавления предложений `if`.

Применение `else` обычно выглядит так:

```
for item in my_list:
    if item.flavor == 'banana':
        break
else:
    raise ValueError('No banana flavor found!')
```

Что касается блоков `try/except`, то на первый взгляд `else` может показаться лишним. Ведь `after_call()` в следующем фрагменте и так будет выполняться, только если `dangerous_call()` не возбудил исключения, верно?

```
try:
    dangerous_call()
    after_call()
except OSError:
    log('OSError...')
```

Однако здесь вызов `after_call()` помещен в блок `try` безо всякой причины. Чтобы код оставался ясным и корректным, в теле блока `try` должны быть только предложения, которые могут возбуждать ожидаемые исключения. Так лучше:

```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

Теперь понятно, что блок `try` защищает от возможных ошибок внутри `dangerous_call()`, но не внутри `after_call()`. Кроме того, явно видно, что `after_call()` выполняется, только если внутри блока `try` не было исключений.

В Python блок `try/except` часто используется для управления потоком выполнения, а не только для обработки ошибок. В официальном глоссарии Python для этого даже есть специальный акроним (*https://docs.python.org/3/glossary.html#term-eafp*):

EAFP

> Проще попросить прощения, чем испрашивать разрешение (Easier to ask for forgiveness than permission). Этот принятый в Python стиль программирования означает следующее: лучше предположить, что ключ или атрибут существует, и перехватить исключение, если предположение окажется неверным. Характерной особенностью этого чистого и быстрого стиля является изобилие предложений try и except. Эта техника противоположна принятому во многих других языках, включая C, стилю LBYL.

Далее в глоссарии определяется акроним LBYL:

LBYL

> Не зная броду, не суйся в воду (Look before you leap). Этот стиль программирования подразумевает проверку предусловий до вызова или поиска. Он противоположен стилю *EAFP* и характеризуется наличием мно-

гочисленных предложений if. В многопоточной программе стиль LBYL чреват состоянием гонки между проверкой и выполнением. Например, код `if key in mapping: return mapping[key]` может привести к ошибке, если другой поток удалит ключ из отображения после проверки, но перед выборкой. Эту проблему можно решить с помощью блокировки или программирования в стиле EAFP.

Принимая во внимание стиль EAFP, использование блоков `else` в предложениях `try/except` выглядит еще более оправданным.

Когда обсуждалось предложение `match`, некоторые (я в том числе) считали, что у него тоже должна быть фраза `else`. Но в итоге было решено, что она ни к чему, потому что `case _:` делает то же самое[1].

Настало время подвести итоги.

Резюме

Мы начали эту главу с рассмотрения контекстных менеджеров и семантики предложения `with`, не ограничиваясь его типичным применением для автоматического закрытия файлов. Мы реализовали свой контекстный менеджер: класс `LookingGlass` с методами `__enter__` и `__exit__`, и показали, как обрабатывать исключения в методе `__exit__`. Ключевой момент, который Раймонд Хэттингер отметил в тезисах к докладу на конференции PyCon US 2013, заключается в том, что блок `with` – это не только средство для управления ресурсами, но и инструмент, позволяющий выделить общий код инициализации и очистки, да и вообще любую пару операций, которые должны быть выполнены до и после какой-то другой процедуры[2].

Мы дали обзор функций в модуле `contextlib` из стандартной библиотеки. Один из них, декоратор `@contextmanager`, дает возможность реализовать контекстный менеджер с помощью простого генератора с одним предложением `yield` – что, конечно, более лаконично, чем кодирование класса, содержащего по меньшей мере два метода. Мы переписали класс `LookingGlass` в виде генераторной функции `looking_glass` и обсудили, как обрабатывать исключения при использовании `@contextmanager`.

Затем изучили элегантный скрипт Питера Норвига *lis.py* – интерпретатор Scheme, написанный на идиоматичном Python и переработанный с использованием `match/case` в функции `evaluate`, составляющей ядро интерпретатора. Для понимания того, как работает `evaluate`, нам пришлось сказать несколько слов о языке Scheme, синтаксическом анализаторе S-выражений, простом цикле REPL

[1] Наблюдая за дискуссией в списке рассылки python-dev, я пришел к выводу, что одной из причин отказа от `else` было отсутствие единого мнения о том, какой должен быть отступ внутри `match`: следует ли помещать `else` на том же уровне, что `match`, или на том же уровне, что `case`?

[2] См. слайд 21 презентации «Python is Awesome» (*https://speakerdeck.com/pyconslides/pycon-keynote-python-is-awesome-by-raymond-hettinger?slide=21*).

и построении вложенных областей видимости с помощью класса `Environment`, наследующего `collection.ChainMap`. А в конце *lis.py* стал для нас полигоном для подробного изучения сопоставления с образцами. Он показывает, как различные части интерпретатора работают совместно, и это помогает лучше понять важные черты самого Python: зачем нужны ключевые слова, как функционируют правила областей видимости, как строятся и используются замыкания.

Дополнительная литература

В главе 8 «Составные предложения» (*https://docs.python.org/3/reference/ compound_stmts.html*) справочного руководства по языку Python имеется все, что можно сказать о части `else` в предложениях `if`, `for`, `while` и `try`. По поводу использования `try/except` в духе Python – с `else` или без – Раймонд Хэттингер дал блестящий ответ на вопрос «Хорошо ли использовать try-except-else в Python?» (*https://stackoverflow.com/questions/16138232/is-it-a-good-practice-to-use-try-except-else-in-python*) на сайте StackOverflow. В книге Алекса Мартелли и др. «Python in a Nutshell», 3-е издание (O'Reilly), имеется глава об исключениях, а в ней – великолепное обсуждение стиля программирования EAFP с отсылкой к одному из пионеров вычислительной техники Грэйс Хоппер, придумавшей фразу: «Проще попросить прощения, чем испрашивать разрешение».

В главе 4 «Встроенные типы» руководства по стандартной библиотеке Python есть раздел, посвященный типам контекстных менеджеров (*http://bit.ly/1MMacTS*). Специальные методы `__enter__` и `__exit__` документированы также в разделе «Контекстные менеджеры и предложение with» справочного руководства по языку Python (*https://docs.python.org/3/library/stdtypes.html#typecontextmanager*). Идея контекстных менеджеров впервые была изложена в документе PEP 343 «The 'with' Statement» (*https://www.python.org/dev/peps/pep-0343/*).

Раймонд Хэттингер в тезисах к докладу на конференции PyCon US 2013 (*https://speakerdeck.com/pyconslides/pycon-keynote-python-is-awesome-by-raymond-hettinger?slide=21*) назвал предложение `with` «призовым средством языка». На той же конференции он продемонстрировал несколько интересных применений контекстных менеджеров в выступлении «Преобразование кода в красивую идиоматичную программу на Python» (*https://speakerdeck.com/pyconslides/transforming-code-into-beautiful-idiomatic-python-by-raymond-hettinger-1?slide=34*).

Статья в блоге Джеффа Прешинга «The Python *with* Statement by Example» (*https://preshing.com/20110920/the-python-with-statement-by-example/*) интересна примерами использования контекстных менеджеров в графической библиотеке `pycairo`.

Класс `contextlib.ExitStack` основан на идее Николауса Рата, который написал короткую статью, объясняющую, чем он полезен: «On the Beauty of Python's ExitStack» (*https://www.rath.org/on-the-beauty-of-pythons-exitstack.html*). Рат пишет, что `ExitStack` похож, но гибче предложения `defer` в Go – на мой взгляд, одной из лучших идей в этом языке.

Бизли и Джонс предлагают контекстные менеджеры для разных целей в своей книге «*Python Cookbook*», 3-е издание (O'Reilly). В рецепте 8.3 «Наделение объектов средствами поддержки протокола управления контекстом» реализован класс `LazyConnection`, экземпляры которого являются контекстными менеджерами, которые автоматически открывают и закрывают сетевое соединение

в блоке `with`. В рецепте 9.22 «Простой способ определения контекстных менеджеров» описываются контекстные менеджеры для хронометража кода транзакционного изменения объекта `list`: в блоке `with` создается копия списка, и все изменения производятся в этой копии. И лишь если блок `with` завершается без исключений, рабочая копия заменяет исходный список. Просто и остроумно.

Питер Норвиг описал свои компактные интерпретаторы языка Scheme в статьях «How to Write a (Lisp) Interpreter (in Python)» (*https://norvig.com/lispy.html*) и «An ((Even Better) Lisp) Interpreter (in Python)» (*https://norvig.com/lispy2.html*). Код *lis.py* и *lispy.py* находится в репозитории *norvig/pytudes* (*https://github.com/norvig/pytudes*). В моем репозитории *fluentpython/lispy* (*https://github.com/fluentpython/lispy*) есть клон *mylis* скрипта *lis.py*, переработанный с учетом новшеств в версии Python 3.10, с улучшенным циклом REPL, интеграцией с командной строкой, примерами, дополнительными тестами и ссылками для желающих ближе познакомиться с языком Scheme. Самый лучший диалект Scheme и среда для изучения и экспериментов – Racket (*https://racket-lang.org/*).

Поговорим

Выделение хлеба из бутерброда

В тезисах к докладу на конференции PyCon US 2013 «What Makes Python Awesome» (*http://pyvideo.org/video/1669/keynote-3*) Раймонд Хэттингер признался, что, впервые увидев предложение по реализации предложения `with`, он счел его «несколько заумным». И у меня поначалу была такая же реакция. Читать PEP'ы зачастую довольно трудно, и PEP 343 в этом отношении типичен.

Но потом, – как сказал нам Хэттингер, – его посетило озарение: подпрограммы – самое важное изобретение в истории языков программирования. Если имеются последовательности операций A;B;C и P;B;Q, то B можно выделить в виде подпрограммы. Это как выделение начинки сэндвича: тунца можно положить на разные куски хлеба. Но что, если требуется выделить сам хлеб и использовать пшеничный хлеб с разными начинками? Именно в этом и состоит смысл предложения `with`. Это дополнение к подпрограммам. Хэттингер продолжает:

> Предложение `with` – могучая штука. Я всем советую не ограничиваться поверхностным знакомством, а копнуть глубже. С его помощью можно делать поразительные вещи. И самые интересные из них еще не открыты. Я полагаю, что если мы сможем найти этому механизму хорошие применения, то он войдет и в другие языки, во все будущие языки. Вы можете принять участие в деянии столь же великом, как изобретение подпрограмм.

Хэттингер признает, что немного перебрал с восхвалением `with`. Тем не менее это действительно очень полезное средство. Когда он воспользовался аналогией с сэндвичем для объяснения того, как `with` дополняет подпрограммы, перед моим мысленным взором возник целый ряд возможностей.

Если вы захотите убедить кого-то в превосходных качествах Python, посмотрите видео тезисов Хэттингера. Часть, относящаяся к контекстным менеджерам, занимает время с 23:00 до 26:15. Но вообще весь материал великолепен.

Эффективная хвостовая рекурсия

В стандартных реализациях Scheme требуется поддержка *чисто хвостовой рекурсии* (proper tail calls – PTC), чтобы замена итерации рекурсией стала практической альтернативой циклам `while` в императивных языках. Некоторые авторы называют PTC *оптимизацией хвостовой рекурсии* (tail call optimization – TCO), для других TCO представляется чем-то совершенно иным. Дополнительные сведения см. в статье Википедии «Tail call» (*https://en.wikipedia.org/wiki/Tail_call*) и в статье «Tail call optimization in ECMAScript 6» (*https://2ality.com/2015/06/tail-call-optimization.html*).

Хвостовой вызов имеет место, когда функция возвращает результат некоторого вызова функции – самой себя или какой-то другой. В примерах функции `gcd` 18.10 и 18.11 рекурсивные хвостовые вызовы производятся в ветви `if`, выполняемой, когда условие ложно.

С другой стороны, в следующей функции `factorial` хвостового вызова нет:

```
def factorial(n):
    if n < 2:
        return 1
    return n * factorial(n - 1)
```

Вызов `factorial` в последней строке не является хвостовым, потому что значение `return` не является результатом рекурсивного вызова: этот результат умножается на `n` перед возвратом.

В примере ниже имеет место хвостовой вызов, поэтому такая функция является хвостово-рекурсивной:

```
def factorial_tc(n, product=1):
    if n < 1:
        return product
    return factorial_tc(n - 1, product * n)
```

В Python нет PTC, поэтому от написания хвостово-рекурсивных функций мы никакого навара не получим. А раз так, то первая версия кажется мне более короткой и понятной. Кстати, при разработке реальных программ имейте в виду, что в Python имеется функция `math.factorial`, написанная на C без всякой рекурсии. Проблема в том, что даже в языках, где PTC реализована, дивиденды получают не все рекурсивные функции, а только специально написанные, так чтобы имел место хвостовой вызов.

Если PTC поддерживается языком, то интерпретатор, видя хвостовой вызов, переходит прямо в тело вызываемой функции, не создавая новый кадр стека, что экономит память. Есть также компилируемые языки, в которых реализована PTC, иногда в качестве оптимизации, включаемой по желанию.

Не существует единого мнения об определении TCO или ценности PTC в языках, которые с самого начала не проектировались как функциональные, например Python или JavaScript. В функциональных языках PTC является ожидаемым свойством, а не просто оптимизацией, которую хорошо бы иметь. Если в языке нет никакого механизма итерации, кроме рекурсии, то PTC необходима из практических соображений. В скрипте Норвига *lis.py* PTC не реализована, но в его же более развитом интерпретаторе *lispy.py* она есть.

Возражения против хвостовой рекурсии в Python и JavaScript

В CPython PTC не реализована и, скорее всего, никогда не будет. Гвидо ван Россум в статье «Final Words on Tail Calls» (*http://neopythonic.blogspot.com/2009/04/final-words-on-tail-calls.html*) объясняет, почему. Приведу главный абзац этой статьи:

> Лично мне кажется, что в некоторых языках это средство представляет ценность, но не думаю, что оно подходит для Python: исключение из трассы стека одних вызовов и оставление других, безусловно, вызовет смятение у многих пользователей, которые не воспитывались в традициях хвостовой рекурсии, но, возможно, постигали семантику вызовов, трассируя их в отладчике.

В 2015 году PTC была включена в стандарт ECMAScript 6 для JavaScript. По состоянию на октябрь 2021 года интерпретатор, входящий в WebKit (*https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/*), ее реализует. WebKit используется в браузере Safari. Интерпретаторы JS во всех остальных основных браузерах обходятся без PTC, как и Node.js, поскольку он опирается на движок V8, который используется в Chrome и сопровождается Google. Транспиляторы и полифилы, транслирующие на JS, например TypeScript, ClojureScript и Babel, тоже не поддерживают PTC, как следует из таблицы совместимости с ECMAScript 6 (*http://kangax.github.io/compat-table/es6/*).

Я встречал несколько объяснений бойкота PTC со стороны разработчиков, но самое распространенное – то, о чем говорил Гвидо ван Россум: PTC усложняет отладку для всех, а преимущества получают только те немногие, кто предпочитает использовать рекурсию вместо итерации. Подробнее см. в статье Грэхема Марлоу «What happened to proper tail calls in JavaScript?» (*https://world.hey.com/mgmarlow/what-happened-to-proper-tail-calls-in-javascript-5494c256*).

Бывают случаи, когда рекурсия является наилучшим решением, даже в Python без PTC. В предыдущей статье (*http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html*) на эту тему Гвидо писал:

> [...] типичная реализация Python допускает глубину рекурсии 1000. Этого вполне хватает для нерекурсивного кода и для кода, который прибегает к рекурсии для обхода, например, типичного дерева синтаксического разбора, но недостаточно, если большой список обходится в рекурсивном цикле.

Я согласен с Гвидо и с большинством разработчиков JS: PTC не годится для Python или JavaScript. Отсутствие PTC – главное препятствие для написания на Python программ в функциональном стиле, более серьезное, чем ограниченный синтаксис `lambda`.

Если вам интересно, как PTC работает в интерпретаторе, менее функционально насыщенном (и более коротком), чем скрипт *lispy.py* Норвига, посмотрите мой скрипт *mylis_2* (*https://github.com/fluentpython/lispy/blob/main/mylis/mylis_2/lis.py*). Хитрость кроется в бесконечном цикле в `evaluate` и в коде в ветви `case` для вызова функций: эта комбинация заставляет интерпретатор перейти в тело следующей `Procedure`, не вызывая `evaluate` рекурсивно в хвостовом вызове. Такие маленькие интерпретаторы демонстрируют мощь абстракции: хотя в самом Python нет PTC, вполне возможно и даже не очень трудно написать на Python интерпретатор, который реализует PTC. Как это делается, я узнал, читая код Питера Норвига. Спасибо, что поделились, профессор!

Замечание Норвига касательно evaluate() с сопоставлением с образцом

Я отправил версию *lis.py* для Python 3.10 Питеру Норвигу. Пример ему понравился, но он предложил другое решение: вместо моих охранных условий он посоветовал иметь ровно одну ветвь `case` для каждого ключевого слова и производить проверки внутри `case`, чтобы выдавать более точные сообщения `SyntaxError` – например, когда тело пусто. Заодно это сделало бы ненужным охранное условие в ветви `case [func_exp, *args] if func_exp not in KEYWORDS:`, потому что каждое ключевое слово обрабатывалось бы раньше, чем `case` для вызовов функций.

Я, наверное, последую совету Норвига, когда буду добавлять функциональность в *mylis*. Но способ, которым я структурировал `evaluate` в примере 18.17, имеет педагогическую ценность именно в этой книге: код легко сравнить с реализацией с помощью `if/elif/…` (пример 2.11), ветви `case` демонстрируют дополнительные возможности сопоставления с образцом, да и короче вышло.

Глава 19

Модели конкурентности в Python

> Предмет конкурентности – как управиться со многими вещами одновременно.
> Предмет параллелизма – как делать много вещей одновременно.
> Не одно и то же, но близко.
> Первое касается структуры, второе – выполнения.
> Конкурентность предлагает способ структурировать решение задачи, которая, возможно (но необязательно), поддается распараллеливанию.
>
> – Роб Пайк, соавтор языка Go[1]

Это глава о том, как заставить Python делать «много вещей одновременно». Для этого может понадобиться конкурентное или параллельное программирование – даже ученые, строго следящие за употреблением терминологии, не согласны в том, как использовать эти термины. Я буду придерживаться неформальных определений Роба Пайка, вынесенных в эпиграф к этой главе, но замечу, что встречал статьи и книги, написанные, по заявлению авторов, о параллельных вычислениях, хотя на самом деле были посвящены в основном конкурентности[2].

С точки зрения Пайка, параллелизм – частный случай конкурентности. Все параллельные системы являются конкурентными, но обратное неверно. В начале 2000-х годов мы пользовались одноядерными машинами, которые могли конкурентно выполнять 100 процессов в GNU Linux. Современный ноутбук с 4 ядрами спокойно выполняет более 200 процессов в каждый момент времени при нормальной повседневной загрузке. Чтобы выполнить 200 задач параллельно, нужно 200 ядер. Поэтому на практике большая часть вычислений производится конкурентно, а не параллельно. ОС управляет сотнями процессов, гарантируя каждому возможность продвигаться вперед, даже если сам процессор может делать не более четырех вещей одновременно.

---

[1] Слайд 8 доклада «Concurrency Is Not Parallelism» (*https://go.dev/blog/waza-talk*).

[2] Я учился, а затем работал с профессором Имре Саймоном, который говаривал, что в науке есть два главных греха: использование разных слов для обозначения одного и того же предмета и использование одного слова для обозначения разных предметов. Имре Саймон (1943–2009) был пионером информатики в Бразилии. Он внес значительный вклад в теорию автоматов и стоял у истоков тропической математики. Он также отстаивал принципы бесплатного программного обеспечения и свободной культуры вообще.

В этой главе у читателя не предполагается предварительных знаний о конкурентном или параллельном программировании. После краткого концептуального введения мы изучим простые примеры, на которых познакомимся с основными пакетами конкурентного программирования в Python – `threading`, `multiprocessing` и `asyncio` – и сравним их.

Последние 30 % главы – общий обзор сторонних инструментов, библиотек, серверов приложений и распределенных очередей задач. Все они могут повысить производительность и масштабируемость Python-приложений. Это важные темы, но они выходят за рамки книги, посвященной базовым возможностям языка Python. Тем не менее я счел необходимым уделить им внимание во втором издании книги, потому что применимость Python в области конкурентных и параллельных вычислений не ограничивается тем, что предлагает стандартная библиотека. Именно поэтому YouTube, DropBox, Instagram, Reddit и другие смогли подстроиться под масштабы веба с момента своего основания, хотя использовали Python в качестве основного языка – вопреки настойчивым заявлениям о том, что «Python не масштабируется».

Что нового в этой главе

Этой главы не было в первом издании. Примеры анимированного индикатора в разделе «Конкурентная программа Hello World» ранее приводились в главе, посвященной *asyncio*. Здесь они улучшены и дают первое представление о трех подходах к конкурентности в Python: потоках, процессах и платформенных сопрограммах.

Весь остальной материал новый, за исключением нескольких абзацев, взятых из прежних глав о `concurrent.futures` и *asyncio*.

Раздел «Python в многоядерном мире» отличается от остальной книги: в нем нет примеров кода. Моей целью было рассказать о важных инструментах, которые вы, возможно, захотите изучить, чтобы достичь такой производительности конкурентности и параллелизма, которая выходит за рамки возможностей стандартной библиотеки Python.

Общая картина

Факторов, осложняющих конкурентное программирование, много, но я хочу остановиться на самом главном: запустить процессы или потоки легко, но как потом отслеживать их[1]?

Когда мы вызываем функцию, вызывающая программа блокируется, пока функция не вернет управление. В этот момент мы знаем, что функция завершила свою работу, и легко можем получить возвращенное значение. Если функция возбуждает исключение, то вызывающая программа может окружить ее операторными скобками `try/except` и перехватить ошибку.

Эти хорошо знакомые действия неприменимы, когда запускается поток или процесс: нет никакого способа автоматически узнать, когда он завершился,

[1] Этот раздел предложил мой друг Брюс Эккель – автор книг о языках Kotlin, Scala, Java, и C++.

а для получения результатов или ошибок нужно организовать какой-то коммуникационный канал, например очередь сообщений.

Кроме того, запуск потока или процесса обходится недешево, поэтому вряд ли стоит это делать, только чтобы выполнить какое-то одно вычисление. Часто мы хотим амортизировать стоимость запуска, сделав поток или процесс «рабочей лошадкой», т. е. исполнителем, который входит в цикл и ждет поступления входных данных, которые нужно обработать. Это еще больше осложняет взаимодействие и ставит новые вопросы. Как заставить исполнителя завершиться, когда необходимость в нем отпала? И как это сделать, не прервав работу на самом интересном месте, в результате чего могли бы остаться недообработанные данные и неосвобожденные ресурсы, например открытые файлы? И снова стандартные ответы подразумевают использование сообщений и очередей.

Запустить сопрограмму несложно. Если для этого использовать ключевое слово `await`, то легко будет получить возвращенное значение, легко отменить сопрограмму, и есть точно определенное место, в котором следует перехватывать исключения. Но сопрограммы обычно запускаются каким-то асинхронным каркасом, поэтому наблюдать за ними так же трудно, как за потоками или процессами.

Наконец, как мы увидим, сопрограммы и потоки не подходят для счетных задач, активно потребляющих процессор.

Поэтому конкурентное программирование требует овладения новыми идеями и приемами кодирования. Но сначала договоримся об основных терминах.

Немного терминологии

Ниже приведены термины, которыми я буду пользоваться в этой и двух последующих главах.

Конкурентность

Способность обрабатывать несколько задач, чередуя выполнение или параллельно (если это возможно), так что каждая задача в конечном итоге успешно доходит до конца или завершается с ошибкой. Одноядерный процессор допускает конкурентность, если работает под управлением планировщика ОС, который чередует выполнение ожидающих задач. Встречается также название многозадачность.

Параллелизм

Способность выполнять несколько вычислений одновременно. Для этого необходим многоядерный процессор, несколько процессоров, графический процессор (GPU) или кластер из нескольких компьютеров.

Единица выполнения

Общий термин для объектов, выполняющих код конкурентно, каждый из которых имеет независимые от других состояния и стек вызовов. Python поддерживает три вида единиц выполнения: *процессы*, *потоки* и *сопрограммы*.

Процесс

Экземпляр компьютерной программы во время ее выполнения, которому выделены память и квант процессорного времени. Современные операционные системы для настольных компьютеров без труда управляют

сотнями конкурентных процессов, при этом каждый процесс изолирован в собственном адресном пространстве. Процессы взаимодействуют посредством каналов, сокетов или отображенных на память файлов – все они могут передавать только «голые» байты. Чтобы передать объект Python из одного процесса в другой, его необходимо сериализовать в виде последовательности байтов. Это дорого, и не все объекты допускают сериализацию. Процесс может порождать подпроцессы, или дочерние процессы. Они изолированы как друг от друга, так и от родительского процесса. Процессы допускают *вытесняющую многозадачность*: планировщик ОС периодически *вытесняет*, т. е. приостанавливает, работающий процесс, чтобы дать возможность поработать остальным. Это означает, что зависший процесс не может подвесить всю систему – теоретически.

Поток

Единица выполнения внутри одного процесса. Сразу после запуска процесс содержит один – главный – поток. Вызывая системные API, процесс может создавать дополнительные потоки, которые будут работать конкурентно. Потоки внутри одного процесса разделяют общее пространство памяти, в которой находятся активные объекты Python. Это позволяет потокам совместно использовать данные, но может приводить к повреждению данных, если сразу несколько потоков пытаются обновить один и тот же объект. Как и процессы, потоки допускают *вытесняющую многозадачность* под управлением планировщика ОС. Поток потребляет меньше ресурсов, чем процесс, для выполнения одной и той же работы.

Сопрограмма

Функция, которая может приостановить свое выполнение и продолжить позже. В Python *классические сопрограммы* строятся на основе генераторных функций, а *платформенные* определяются с помощью ключевых слов `async def`. В разделе «Классические сопрограммы» главы 17 это понятие было определено, а в главе 21 будет рассмотрено использование платформенных сопрограмм. В Python сопрограммы обычно исполняются в одном потоке под управлением *цикла событий*, который работает в том же потоке. Такие каркасы асинхронного программирования, как *asyncio*, *Curio* или *Trio*, предоставляют цикл событий и поддерживающие библиотеки для реализации неблокирующего ввода-вывода на основе сопрограмм. Сопрограммы поддерживают *кооперативную многозадачность*: каждая сопрограмма должна явно уступать процессор с помощью ключевого слова `yield` или `await`, чтобы другие части программы могли работать конкурентно (но не параллельно). Это означает, что любой блокирующий код внутри сопрограммы блокирует выполнение цикла событий и всех остальных сопрограмм – в отличие от *вытесняющей многозадачности*, которую поддерживают процессы и потоки. С другой стороны, сопрограммы потребляют меньше ресурсов по сравнению с процессами и потоками, выполняющими ту же работу.

Очередь

Структура данных, позволяющая помещать и извлекать элементы, обычно в порядке FIFO: первым пришел, первым ушел. Очереди дают возможность

единицам выполнения обмениваться данными и управляющими сообщениями, например кодами ошибок и сигналами завершения. Реализация очереди зависит от модели конкурентности: пакет `queue` в стандартной библиотеке Python предоставляет классы очередей для поддержки потоков, тогда как пакеты `multiprocessing` и `asyncio` реализуют собственные классы очередей. Пакеты `queue` и `asyncio` включают также очереди, обслуживаемые не в порядке FIFO: `LifoQueue` и `PriorityQueue`.

Блокировка

Объект, который единицы выполнения могут использовать для синхронизации своих действий, чтобы избежать повреждения данных. Во время обновления разделяемой структуры данных исполняемый код должен удерживать ассоциированную блокировку. Это служит для остальных частей программы сигналом, что нужно подождать, пока блокировка освободится, и только потом обращаться к той же структуре данных. Простейший вид блокировки называется мьютексом (mutual exclusion – взаимное исключение). Реализация блокировки зависит от модели конкурентности.

Состязание

Спор за ограниченный ресурс. Состязание возникает, когда несколько единиц выполнения пытаются обратиться к разделяемому ресурсу, например блокировке или хранилищу. Бывает также состязание за процессор, когда счетные процессы или потоки должны ждать, пока планировщик ОС выделит им долю процессорного времени.

Теперь воспользуемся введенной терминологией, чтобы понять, как конкурентность поддерживается в Python.

Процессы, потоки и знаменитая блокировка GIL в Python

Ниже описано, как только что рассмотренные понятия применяются в контексте программирования на Python.

1. Каждый экземпляр интерпретатора Python является процессом. Дополнительные процессы Python можно запускать с помощью библиотек *multiprocessing* или *concurrent.futures*. Библиотека *subprocess* предназначена для запуска процессов, в которых будут исполняться внешние программы, написанные на любом языке.
2. Интерпретатор Python использует единственный поток, в котором выполняется и пользовательская программа, и сборщик мусора. Для запуска дополнительных потоков предназначены библиотеки *threading* и *concurrent.futures*.
3. Доступ к счетчикам ссылок на объекты и другим внутренним структурам интерпретатора контролируется глобальной блокировкой интерпретатора (Global Interpreter Lock – GIL). Только один поток Python может удерживать GIL в каждый момент времени. Это означает, что только один поток может выполнять Python-код, и от числа процессорных ядер это не зависит.

4. Чтобы помешать потоку Python удерживать GIL бесконечно, интерпретатор байт-кода Python периодически (по умолчанию раз в 5 миллисекунд[1]) приостанавливает текущий поток и тем самым освобождает GIL. Поток может попытаться снова захватить GIL, но если его ждут другие потоки, то планировщик ОС, возможно, выберет один из них.
5. Программист, пишущий на Python, не может управлять GIL. Но встроенная функция или расширение, написанное на C или на любом другом языке, имеющем интерфейс к Python на уровне C API, может освободить GIL во время выполнения длительной задачи.
6. Любая стандартная библиотечная функция Python, делающая системный вызов[2], освобождает GIL. Сюда относятся все функции, выполняющие дисковый ввод-вывод, сетевой ввод-вывод, а также `time.sleep()`. Многие счетные функции в библиотеках NumPy/SciPy, а также функции сжатия и распаковки из модулей `zlib` и `bz2` также освобождают GIL[3].
7. Расширения, интегрированные на уровне интерфейса между Python и C, могут тоже запускать потоки, не управляемые Python, на которые действие GIL не распространяется. Такие свободные от GIL потоки в общем случае не могут изменять объекты Python, но могут читать и записывать память объектов, поддерживающих протокол буфера, например `bytearray`, `array.array` и массивы *NumPy*.
8. Влияние GIL на сетевое программирование с помощью потоков Python сравнительно невелико, потому что функции ввода-вывода освобождают GIL, а чтение или запись в сеть всегда подразумевает высокую задержку по сравнению с чтением-записью в память. Следовательно, каждый отдельный поток все равно тратит много времени на ожидание, так что их выполнение можно чередовать без заметного снижения общей пропускной способности. Потому-то Дэвид Бизли и говорил: «Python отлично умеет ничего не делать»[4].
9. Состязание за GIL замедляет работу счетных потоков в Python. В таких случаях последовательный однопоточный код проще и быстрее.
10. Для выполнения счетного Python-кода на нескольких ядрах нужно использовать несколько процессов Python.

Процитируем удачное резюме из документации по модулю `threading`[5]:

---

[1] Функция `sys.getswitchinterval()` возвращает текущее значение интервала, а функция `sys.setswitchinterval(s)` изменяет его.

[2] Системным вызовом называется обращение из пользовательского кода к функции, находящейся в ядре операционной системы. Ввод-вывод, таймеры и блокировки – примеры служб ядра, доступных через системные вызовы. Дополнительные сведения можно почерпнуть из статьи Википедии «System call» (*https://en.wikipedia.org/wiki/System_call*).

[3] Модули `zlib` и `bz2` специально отмечены в сообщении Антуана Питру (он добавил GIL с квантованием времени в Python 3.2) в списке рассылки python-dev (*https://mail.python.org/pipermail/python-dev/2009-October/093356.html*).

[4] Источник: слайд 106 пособия Бизли «Generators: The Final Frontier» (*http://www.dabeaz.com/finalgenerator/*).

[5] Источник: последний абзац раздела «Объекты потоков» (*https://docs.python.org/3/library/threading.html#thread-objects*).

> **Деталь реализации CPython.** В CPython, из-за глобальной блокировки интерпретатора, в каждый момент времени Python-код может выполняться только одним потоком (хотя некоторые высокопроизводительные библиотеки умеют обходить это ограничение). Если вы хотите, чтобы приложение более эффективно использовало вычислительные ресурсы многоядерных машин, то пользуйтесь модулем `multiprocessing` или классом `concurrent.futures.ProcessPoolExecutor`. Однако многопоточное выполнение все же является вполне пригодной моделью, если требуется одновременно выполнять несколько задач с большим объемом ввода-вывода.

Этот пассаж начинается словами «Деталь реализации CPython», потому что GIL не является частью определения языка Python. В реализациях Jython и IronPython нет GIL. К сожалению, обе они сильно отстают – все еще остаются на уровне Python 2.7. В высокопроизводительном интерпретаторе PyPy (*https://www.pypy.org/*) GIL тоже имеется – в версиях 2.7 и 3.7 (последняя датирована июнем 2021 года).

В этом разделе не упоминаются сопрограммы, потому что по умолчанию все они вкупе с управляющим циклом событий, который предоставлен каркасом асинхронного программирования, работают в одном потоке, поэтому GIL не оказывает на них никакого влияния. Можно использовать несколько потоков в асинхронной программе, но рекомендуется, чтобы и цикл событий, и все сопрограммы исполнялись в одном потоке, а дополнительные потоки выделять для специальных задач. Мы объясним эту идею в разделе «Делегирование задач исполнителям» главы 21.

Но хватит пока теории. Посмотрим на код.

Конкурентная программа Hello World

Говоря о потоках и о том, как избежать GIL, соразработчик Python Мишель Симионато опубликовал пример (*https://mail.python.org/pipermail/python-list/2009-February/675659.html*), похожий на конкурентную «Hello World» – простейшую программу, показывающую, как Python может «идти и одновременно жевать резинку».

В программе Симионато используется модуль `multiprocessing`, а я дополнительно адаптировал ее под `threading` и `asyncio`. Начнем с версии для `threading`, которая может показаться знакомой тем, кто изучал потоки в Java или C.

Анимированный индикатор с потоками

Идея следующих далее примеров проста: запустить функцию, которая блокирует выполнение на 3 секунды, пока чередует символы на экране терминала, давая пользователю понять, что программа «думает», а не зависла.

Скрипт выводит анимированный индикатор, т. е. отображает символы из строки `«\|/-»` в одной и той же позиции экрана[1]. По завершении медленного вычисления строка индикатора очищается и выводится результат: `Answer: 42`.

[1] В Unicode немало символов, полезных для простых анимаций, например знаки алфавита Брайля. Я использовал ASCII-символы `"\|/-"`, чтобы не усложнять пример.

На рис. 19.1 показано, что выводят две версии примера: первая с потоками, вторая с сопрограммами. Если вы далеко от компьютера, то представьте себе, что знак \ в последней строке крутится.

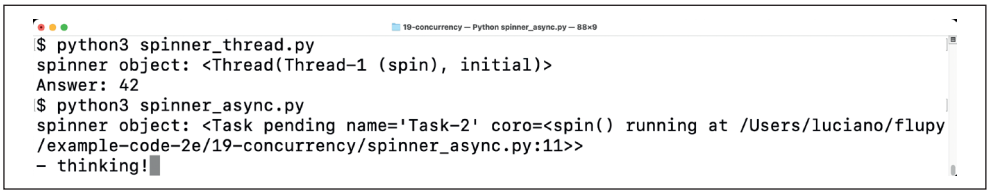


Рис. 19.1. Скрипты spinner_thread.py и spinner_asyncio.py порождают похожие результаты: repr-представление объекта `spinner` и текст `Answer: 42`. На снимке экрана скрипт spinner_asyncio.py еще работает, поэтому отображается сообщение `\ thinking!`; когда скрипт завершится, эту строку заменит `Answer: 42`

Сначала рассмотрим скрипт *spinner_thread.py*. В примере 19.1 показаны первые две функции скрипта, а в примере 19.2 все остальное.

Пример 19.1. spinner_thread.py: функции `spin` и `slow`

```
import itertools
import time
from threading import Thread, Event

def spin(msg: str, done: Event) -> None: ❶
    for char in itertools.cycle(r'\|/-'): ❷
        status = f'\r{char} {msg}' ❸
        print(status, end='', flush=True)
        if done.wait(.1): ❹
            break ❺
        blanks = ' ' * len(status)
        print(f'\r{blanks}\r', end='') ❻

def slow() -> int:
    time.sleep(3) ❼
    return 42
```

❶ Эта функция будет работать в отдельном потоке. Аргумент `done`, экземпляр класса `threading.Event`, – простой объект для синхронизации потоков.

❷ Это бесконечный цикл, потому что `itertools.cycle` отдает по одному символу за раз и перебирает заданную строку по кругу.

❸ Хитрость, позволяющая выполнить анимацию в текстовом режиме: возвращаем курсор в начало строки, печатая управляющий символ возврата каретки (`'\r'`).

❹ Метод `Event.wait(timeout=None)` возвращает `True`, когда другой поток установил событие; если же истек тайм-аут `timeout`, то он возвращает `False`. Тайм-аут `.1s` означает, что анимация производится с частотой 10 кадров в секунду. Чтобы индикатор крутился быстрее, задайте тайм-аут поменьше.

❺ Выйти из бесконечного цикла.

❻ Очистить строку состояния, затирая ее пробелами и возвращая курсор в начало строки.

❼ Функция `slow()` вызывается из главного потока. Представьте, что это вызов медленного API по сети. Вызов `sleep` блокирует главный поток, но GIL при этом освобождается, поэтому поток индикатора продолжает работать.

В этом примере нужно обратить внимание на то, что `time.sleep()` блокирует вызывающий поток, но освобождает GIL, позволяя работать другим потокам Python.

Функции `spin` и `slow` будут выполняться параллельно. Главный поток – единственный существующий в начале работы программы – запускает новый поток, исполняющий `spin`, а затем вызывает `slow`. В Python сознательно не предусмотрен API для завершения потока. Чтобы остановить поток, ему необходимо отправить сообщение.

Класс `threading.Event` – самый простой из имеющихся в Python механизмов сигнализации для координации потоков. В экземпляре `Event` имеется внутренний булев флаг, который первоначально равен `False`. Вызов `Event.set()` устанавливает этот флаг в `True`. Если флаг равен `False`, то поток, вызвавший `Event.wait()`, блокируется до тех пор, пока какой-нибудь другой поток не вызовет `Event.set()`, и в этот момент `Event.wait()` возвращает `True`. Если функции `Event.wait(s)` передан тайм-аут в секундах, то по истечении тайм-аута этот вызов вернет `False` (или `True`, если раньше какой-то другой поток вызовет `Event.set()`).

Функция `supervisor` в примере 19.2 использует `Event` как сигнал о том, что `spin` должна закончить работу.

Пример 19.2. spinner_thread.py: функции `supervisor` и `main`

```
def supervisor() -> int: ❶
    done = Event() ❷
    spinner = Thread(target=spin, args=('thinking!', done)) ❸
    print(f'spinner object: {spinner}') ❹
    spinner.start() ❺
    result = slow() ❻
    done.set() ❼
    spinner.join() ❽
    return result

def main() -> None:
    result = supervisor() ❾
    print(f'Answer: {result}')

if __name__ == '__main__':
    main()
```

❶ `supervisor` возвращает результат `slow`.

❷ Экземпляр `threading.Event` – ключ к координации потоков `main` и `spinner`, как будет объяснено ниже.

❸ Чтобы создать новый экземпляр `Thread`, задайте функцию в именованном аргументе `target`, а необходимые ей позиционные аргументы передавайте в кортеже `args`.

❹ Отобразить объект `spinner`. Результатом будет представление `<Thread(Thread-1, initial)>`, где `initial` – состояние потока, означающее, что он еще не запущен.
❺ Запустить поток `spinner`.
❻ Вызвать функцию `slow`, которая блокирует поток `main`. Тем временем второй поток выполняет анимацию индикатора.
❼ Установить флаг `Event` в `True`; в результате произойдет выход из цикла `for` в функции `spin`.
❽ Ждать завершения потока `spinner`.
❾ Вызвать функцию `supervisor`. Я разделил функции `main` и `supervisor`, чтобы этот пример больше походил на версию с `asyncio` в примере 19.4.

Когда поток `main` устанавливает событие `done`, поток `spinner` замечает это и завершается чисто.

Теперь рассмотрим аналогичный пример с использованием пакета `multiprocessing`.

Индикатор с процессами

Пакет `multiprocessing` поддерживает выполнение конкурентных задач в отдельных процессах Python вместо потоков. После создания экземпляра `multiprocessing.Process` в дочернем процессе запускается новый интерпретатор Python, работающий в фоновом режиме. Так как у каждого процесса Python свой собственный GIL, это дает программе потенциальную возможность использовать все доступные процессорные ядра, но будет ли ей это позволено, зависит от планировщика операционной системы. Практические последствия мы увидим в разделе «Доморощенный пул процессов» ниже, но для нашей простой программы всё это не имеет значения.

Цель этого раздела – познакомиться с пакетом `multiprocessing` и показать, что его API эмулирует API `threading`, что упрощает переход от потоков к процессам в простых программах, как показано в скрипте *spinner_proc.py* (пример 19.3).

Пример 19.3. spinner_proc.py: показаны только изменившиеся части; весь остальной код такой же, как в spinner_thread.py

```
import itertools
import time
from multiprocessing import Process, Event ❶
from multiprocessing import synchronize ❷

def spin(msg: str, done: synchronize.Event) -> None: ❸

# [опущено] функции spin и slow не изменились по сравнению со spinner_thread.py
def supervisor() -> int:
    done = Event()
    spinner = Process(target=spin, ❹
                      args=('thinking!', done))
    print(f'spinner object: {spinner}') ❺
    spinner.start()
    result = slow()
    done.set()
    spinner.join()
    return result

# [опущено] функция main тоже не изменилась
```

❶ Базовый API `multiprocessing` имитирует API `threading`, но аннотации типов и Муру выявляют различие: `multiprocessing.Event` – функция (а не класс, как `threading.Event`), которая возвращает `synchronize.Event` …
❷ … что вынуждает нас импортировать `multiprocessing.synchronize` …
❸ … чтобы записать эту аннотацию типа.
❹ Простое использование класса `Process` похоже на `Thread`.
❻ Объект `spinner` отображается как `<Process name='Process-1' parent=14868 initial>`, где `14868` – идентификатор процесса Python, в котором исполняется скрипт *spinner_proc.py*.

Базовые API пакетов `threading` и `multiprocessing` похожи, но их реализации сильно различаются, а API `multiprocessing` гораздо обширнее, что отражает сложность многопроцессного программирования. Например, одна из сложностей, возникающих при переходе от потоков к процессам, – как обеспечить взаимодействие процессов, которые изолированы операционной системой и не могут разделять объекты Python. Это означает, что объекты, пересекающие границы процессов, необходимо сериализовывать и десериализовывать, неся на это дополнительные затраты. В примере 19.3 границу процессов пересекает только состояние `Event`, которое реализовано низкоуровневым семафором на уровне ОС в написанном на C коде модуля `multiprocessing`[1].

Начиная с версии Python 3.8 в стандартной библиотеке имеется пакет `multiprocessing.shared_memory`, но он не поддерживает экземпляры пользовательских классов. Помимо необработанных байтов, этот пакет позволяет процессам разделять объекты `ShareableList` – это тип изменяемой последовательности, который может содержать фиксированное число элементов типа `int`, `float`, `bool` и `None`, а также `str` и `bytes` размером не более 10 МБ каждый. Дополнительные сведения см. в документации по классу `ShareableList` (*https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.shared_memory.ShareableList*).

Теперь посмотрим, как такого же поведения можно достичь с помощью сопрограмм вместо потоков и процессов.

Индикатор с сопрограммами

Глава 21 целиком посвящена асинхронному программированию с использованием сопрограмм. Здесь же приводится только общее введение, позволяющее сравнить этот подход с моделями конкурентности на основе потоков и процессов. Поэтому многие детали опущены.

Выделение процессорного времени потокам и процессам – задача планировщика ОС. С другой стороны, сопрограммы приводятся в действие циклом со-

[1] Семафор – это фундаментальный строительный блок, который можно использовать для реализации механизмов синхронизации. Python предлагает различные классы семафоров для использования совместно с потоками, процессами и сопрограммами. С классом `asyncio.Semaphore` мы познакомимся в разделе «Использование asyncio.as_completed и Thread» главы 21.

бытий, находящимся на уровне приложения. Он управляет очередью ожидающих активации сопрограмм, выполняет их по одной, отслеживает события, генерируемые операциями ввода-вывода, инициированными сопрограммами, и возвращает управление соответствующей сопрограмме, когда такое событие происходит. Цикл событий, библиотечные и пользовательские сопрограммы выполняются в одном потоке. Поэтому когда какая-то сопрограмма расходует время, замедляется цикл событий – и все остальные сопрограммы.

Вариант программы индикатора с сопрограммами будет проще понять, если начать с функции `main`, а затем перейти к `supervisor`. Обе показаны в примере 19.4.

Пример 19.4. spinner_async.py: функция `main` и сопрограмма `supervisor`

```
def main() -> None: ❶
    result = asyncio.run(supervisor()) ❷
    print(f'Answer: {result}')

async def supervisor() -> int: ❸
    spinner = asyncio.create_task(spin('thinking!')) ❹
    print(f'spinner object: {spinner}') ❺
    result = await slow() ❻
    spinner.cancel() ❼
    return result

if __name__ == '__main__':
    main()
```

❶ `main` – единственная настоящая функция в этой программе, все остальные – сопрограммы.

❷ Функция `asyncio.run` запускает цикл событий, активирующий сопрограмму, которая в конечном итоге приведет в действие и другие сопрограммы. Функция `main` остается блокированной, пока `supervisor` не вернет управление. Значение, возвращенное `supervisor`, станет значением, возвращенным `asyncio.run`.

❸ Платформенные сопрограммы определяются с помощью ключевых слов `async def`.

❹ `asyncio.create_task` планирует выполнение `spin` сразу после возврата экземпляра `asyncio.Task`.

❺ Представление `repr` объекта `spinner` имеет вид `<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:11>>`.

❻ Ключевое слово `await` вызывает `slow`, блокируя `supervisor` до возврата из `slow`. Значение, возвращенное `slow`, присваивается переменной `result`.

❼ Метод `Task.cancel` возбуждает исключение `CancelledError` внутри сопрограммы `spin`, как мы увидим в примере 19.5.

В примере 19.4 демонстрируется три основных способа выполнить сопрограмму:

`asyncio.run(coro())`

Вызывается из регулярной функции для управления объектом сопрограммы, который обычно является точкой входа в весь асинхронный код программы, как `supervisor` в этом примере. Этот вызов блокирует выполнение,

пока `coro` не вернет управление. Функция `run()` возвращает значение, возвращенное `coro`.

`asyncio.create_task(coro())`

Вызывается из сопрограммы, чтобы запланировать выполнение другой сопрограммы. Этот вызов не приостанавливает текущую сопрограмму. Он возвращает экземпляр `Task` – объект, который обертывает объект сопрограммы и предоставляет методы для управления ей и опроса ее состояния.

`await coro()`

Вызывается из сопрограммы, чтобы передать управление объекту сопрограммы, возвращенному `coro()`. Этот вызов приостанавливает текущую сопрограмму до возврата из `coro`. Значением выражения `await` является значение, возвращенное `coro`.

Запомните: вызов сопрограммы как `coro()` сразу же возвращает объект сопрограммы, но не выполняет тело функции `coro`. Активация тел сопрограмм – задача цикла событий.

Теперь рассмотрим сопрограммы `spin` и `slow` в примере 19.5.

Пример 19.5. spinner_async.py: сопрограммы `spin` и `slow`

```
import asyncio
import itertools

async def spin(msg: str) -> None: ❶
    for char in itertools.cycle(r'\|/-'):
        status = f'\r{char} {msg}'
        print(status, flush=True, end='')
        try:
            await asyncio.sleep(.1) ❷
        except asyncio.CancelledError: ❸
            break
    blanks = ' ' * len(status)
    print(f'\r{blanks}\r', end='')

async def slow() -> int:
    await asyncio.sleep(3) ❹
    return 42
```

❶ Нам не нужен аргумент `Event`, который в *spinner_thread.py* (пример 19.1) использовался, чтобы просигнализировать о завершении `slow`.

❷ Использовать `await asyncio.sleep(.1)` вместо `time.sleep(.1)`, чтобы приостановить выполнение без блокировки других сопрограмм. См. эксперимент после этого примера.

❸ Когда вызывается метод `cancel` объекта `Task`, управляющего этой сопрограммой, возбуждается исключение `asyncio.CancelledError`. Время выходить из цикла.

❹ Сопрограмма `slow` также использует `await asyncio.sleep` вместо `time.sleep`.

Эксперимент: ломаем индикатор ради озарения

Я рекомендую провести следующий эксперимент, чтобы понять, как работает *spinner_async.py*. Импортируйте модуль `time`, затем перейдите к сопрограмме `slow` и замените строку `await asyncio.sleep(3)` вызовом `time.sleep(3)`, как в примере 19.6.

Пример 19.6. spinner_async.py: замена `await asyncio.sleep(3)` на `time.sleep(3)`

```
async def slow() -> int:
    time.sleep(3)
    return 42
```

Наблюдение за поведением лучше запоминается, чем чтение текста. Давайте, я подожду.

Вот что вы увидите при проведении эксперимента.

1. Отображается представление объекта индикатора в виде `<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:12>>`.
2. Сам индикатор так и не появляется. Программа висит 3 секунды.
3. Отображается сообщение `Answer: 42`, и программа завершается.

Чтобы понять, что происходит, вспомним, что в Python-коде, где используется `asyncio`, есть лишь один поток выполнения, если только явно не запущены дополнительные потоки или процессы. Это значит, что в каждый момент времени выполняется всего одна сопрограмма. Конкурентность достигается путем передачи управления от одной сопрограммы к другой. В примере 19.7 мы сосредоточимся на том, что происходит в сопрограммах `supervisor` и `slow` в предложенном эксперименте.

Пример 19.7. spinner_async_experiment.py: сопрограммы `supervisor` и `slow`

```
async def slow() -> int:
    time.sleep(3)
    return 42 ❹

async def supervisor() -> int:
    spinner = asyncio.create_task(spin('thinking!')) ❶
    print(f'spinner object: {spinner}') ❷
    result = await slow() ❸
    spinner.cancel() ❺
    return result
```

❶ Создается задача `spinner`, чтобы в конечном итоге активировать выполнение `spin`.

❷ На экране показано, что `Task` находится в состоянии «pending».

❸ Выражение `await` передает управление сопрограмме `slow`.

❹ `time.sleep(3)` блокирует выполнение на 3 секунды; в программе ничего не может произойти, потому что главный поток блокирован, а он единственный. Операционная система продолжит заниматься другими делами. Спустя 3 секунды `sleep` завершается, выполнение возобновляется и `slow` возвращает управление.

❺ Сразу после возврата из `slow` задача `spinner` отменяется. Поток управления так и не дошел до тела сопрограммы `spin`.

Скрипт *spinner_async_experiment.py* преподал нам важный урок, который объясняется в следующем предупреждении.

Никогда не используйте `time.sleep(…)` в сопрограммах `asyncio`, если не хотите приостановить всю программу в целом. Если сопрограмма хочет потратить некоторое время, ничего не делая, она должна вызвать `await asyncio.sleep(DELAY)`. Так она уступит управление циклу событий `asyncio`, который может дать поработать другим ожидающим сопрограммам.

Greenlet и gevent

Говоря об организации конкурентности на основе сопрограмм, важно упомянуть пакет *greenlet* (*https://greenlet.readthedocs.io/en/latest/*), существующий уже много лет и активно используемый[1]. Этот пакет поддерживает кооперативную многозадачность с помощью облегченных сопрограмм – *гринлетов*, – которые не требуют специального синтаксиса типа `yield` или `await`, а потому проще интегрируются с уже имеющимися последовательными кодовыми базами. В каркасе объектно-реляционных отображений SQL Alchemy 1.4 гринлеты используются на внутреннем уровне (*https://docs.sqlalchemy.org/en/14/changelog/migration_14.html#asynchronous-io-support-for-core-and-orm*) для реализации нового асинхронного API (*https://docs.sqlalchemy.org/en/14/orm/extensions/asyncio.html),* совместимого с *asyncio*.

Сетевая библиотека *gevent* (*http://www.gevent.org/*) занимается партизанским латанием стандартного модуля `socket` и делает его неблокирующим путем замены части кода гринлетами. Эта библиотека в значительной степени прозрачна для окружающего кода, что упрощает адаптацию последовательных приложений и библиотек (например, драйверов баз данных) к выполнению конкурентного сетевого ввода-вывода. *gevent* применяется во многих проектах с открытым исходным кодом (*https://github.com/gevent/gevent/wiki/Projects*), в т. ч. в широко распространенном проекте *Gunicorn* (*https://gunicorn.org/*), упомянутом в разделе «Серверы приложений WSGI» ниже.

Сравнение супервизоров

Количество строк в скриптах *spinner_thread.py* и *spinner_async.py* почти одинаково. Главным в этих примерах являются функции `supervisor`. Давайте сравним их. В примере 19.8 показана только функция `supervisor` из примера 19.2.

Пример 19.8. spinner_thread.py: многопоточная функция `supervisor`

```
def supervisor() -> int:
    done = Event()
    spinner = Thread(target=spin,
                     args=('thinking!', done))
    print('spinner object:', spinner)
    spinner.start()
```

[1] Спасибо техническим рецензентам Калебу Хэттингу и Юргену Гмаху, не давшим мне пройти мимо greenlet и gevent.

```
    result = slow()
    done.set()
    spinner.join()
    return result
```

Для сравнения в примере 19.9 показана сопрограмма `supervisor` из примера 19.4.

Пример 19.9. spinner_async.py: асинхронная сопрограмма `supervisor`

```
async def supervisor() -> int:
    spinner = asyncio.create_task(spin('thinking!'))
    print('spinner object:', spinner)
    result = await slow()
    spinner.cancel()
    return result
```

Перечислим заслуживающие внимания сходства и различия между обеими реализациями `supervisor`:

- класс `asyncio.Task` приблизительно эквивалентен `threading.Thread`;
- `Task` управляет объектом сопрограммы, а `Thread` обращается к вызываемому объекту;
- сопрограмма уступает управление явно с помощью ключевого слова `await`;
- мы не создаем объекты `Task` самостоятельно, а получаем их, передавая сопрограмму функции `asyncio.create_task(…)`;
- когда `asyncio.create_task(…)` возвращает объект `Task`, его выполнение уже запланировано, тогда как экземпляру `Thread` нужно явно сказать, что пора выполняться, вызвав его метод `start`;
- в многопоточной версии `supervisor` `slow` является простой функцией и непосредственно вызывается из главного потока. В асинхронной же версии `slow` – сопрограмма, активируемая `await`;
- не существует API для завершения потока извне, вместо этого нужно послать потоку сигнал, например установить объект `Event` `done`. Для задач существует метод экземпляра `Task.cancel()`, который возбуждает исключение `CancelledError` в том выражении `await`, в котором в настоящий момент приостановлено выполнение тела сопрограммы;
- сопрограмму `supervisor` нужно запускать с помощью `asyncio.run` в функции `main`.

Это сравнение поможет вам понять, как *asyncio* координирует конкурентные задания и чем его подход отличается от принятого в модуле `Threading`, с которым вы, возможно, лучше знакомы.

И последний момент, отличающий потоки от сопрограмм: если вы когда-нибудь писали нетривиальную программу с потоками, то знаете, как трудно рассуждать о поведении программы, которую планировщик может прервать в любое время. Нужно не забыть поставить блокировки, защищающие критические секции программы, чтобы ее не прервали в середине многошаговой операции, что могло бы оставить данные в некорректном состоянии.

В случае сопрограмм код по умолчанию защищен от прерывания. Вы должны явно выполнить `await`, чтобы другие части программы могли поработать. Вместо синхронизации потоков с помощью блокировок сопрограммы «син-

хронизированы» по определению: в каждый момент времени может работать только одна. Желая добровольно отказаться от владения процессором, мы используем `await`, чтобы уступить управление планировщику. Именно поэтому сопрограмму можно безопасно отменить: по определению, сопрограмма может быть отменена только тогда, когда приостановлена в выражении `await`, и ничто не мешает произвести очистку, обработав исключение `CancelledError`.

Вызов `time.sleep()` блокирует выполнение, но ничего не делает. В следующем эксперименте мы рассмотрим вызов счетной функции, чтобы лучше понять GIL, а также оценить влияние счетных функций на асинхронный код.

Истинное влияние GIL

В многопоточном коде (пример 19.1) вызов `time.sleep(3)` в функции `slow` можно заменить клиентским HTTP-запросом из какой-нибудь библиотеки, и индикатор продолжит крутиться. Объясняется это тем, что правильно спроектированная сетевая библиотека освобождает GIL на время ожидания ответа из сети.

Выражение `asyncio.sleep(3)` в сопрограмме `slow` можно заменить выражением `await`, ожидающим ответа от хорошо спроектированной сетевой библиотеки, потому что такие библиотеки предоставляют сопрограммы, уступающие управление циклу событий на время ожидания ответа. А тем временем индикатор продолжает крутиться.

В случае счетного кода все обстоит иначе. Рассмотрим функцию `is_prime` в примере 19.10, которая возвращает `True`, если аргумент – простое число, а иначе `False`.

Пример 19.10. primes.py: элементарная проверка на простоту из примера ProcessPoolExecutor в документации по Python (*https://docs.python.org/3/library/ concurrent.futures.html#processpoolexecutor-example*)

```
def is_prime(n: int) -> bool:
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    root = math.isqrt(n)
    for i in range(3, root + 1, 2):
        if n % i == 0:
            return False
    return True
```

Вызов `is_prime(5_000_111_000_222_021)` занимает примерно 3.3 с на корпоративном ноутбуке, которым я сейчас пользуюсь[1].

Проверка знаний

Вспомните все, чему вы научились, и попробуйте ответить на следующий вопрос, состоящий из трех частей. Одна часть трудная (по крайней мере, оказалась такой для меня).

[1] MacBook Pro 2018 с 15-дюймовым экраном и 6-ядерным процессором Intel Core i7 2.2 ГГц.

Что произойдет с анимацией индикатора, если произвести следующие изменения в предположении, что `n = 5_000_111_000_222_021` – простое число, для проверки которого на моей машине потребовалось 3.3 с:

1. В *spinner_proc.py* заменить `time.sleep(3)` вызовом `is_prime(n)`?
2. В *spinner_thread.py* заменить `time.sleep(3)` вызовом `is_prime(n)`?
3. В *spinner_async.py* заменить `await asyncio.sleep(3)` вызовом `is_prime(n)`?

Прежде чем выполнять код или читать дальше, попробуйте найти ответы самостоятельно. А потом можете скопировать скрипты *spinner_*.py* и модифицировать их, как предложено.

А теперь ответы, от самого простого к самому трудному.

1. Ответ для multiprocessing

Индикатор управляется дочерним процессом, поэтому будет крутиться и тогда, когда родительский процесс проверяет число на простоту[1].

2. Ответ для threading

Индикатор управляется дополнительным потоком, поэтому будет крутиться и тогда, когда главный поток проверяет число на простоту.

Я пришел к этому ответу не сразу: я ожидал, что индикатор прекратит крутиться, потому что переоценил воздействие GIL.

В этом примере индикатор продолжает крутиться, потому что Python приостанавливает работающий поток раз в 5 мс (по умолчанию), делая GIL доступной другим ожидающим потокам. Поэтому главный поток, исполняющий `is_prime`, прерывается каждые 5 мс, так что у дополнительного потока есть возможность проснуться и выполнить одну итерацию цикла `for`, в конце которой он вызовет метод `wait` события `done` и освободит GIL. Затем главный поток захватит GIL и вычисление `is_prime` продолжится на протяжении следующих 5 мс.

Это не оказывает видимого влияния на время работы этого конкретного примера, поскольку функция `spin` быстро выполняет одну итерацию и освобождает GIL в ожидании события `done`, поэтому интенсивность состязания за GIL невелика. Главный поток, исполняющий `is_prime`, владеет GIL большую часть времени.

В этом простом примере мы разобрались со счетной задачей, потому что потоков всего два: один на всю катушку загружает процессор, а второй просыпается жалкие 10 раз в секунду, чтобы обновить индикатор.

Но если потоков два или больше и все они алчно потребляют процессорное время, то программа будет работать медленнее, чем последовательный код.

3. Ответ для asyncio

Если вызвать `is_prime(5_000_111_000_222_021)` в сопрограмме `slow` в примере *spinner_async.py*, то индикатор вообще не появится на экране. Эффект бу-

[1] В наши дни это так, потому что вы, скорее всего, пользуетесь современной ОС с вытесняющей многозадачностью. Но Windows до эры NT и macOS до эры OS X не были «вытесняющими», поэтому любой процесс мог захватить процессор на 100 %, в результате чего вся система подвисала. Даже сегодня мы не совсем ушли от этой проблемы, но поверьте старику: в 1990-х она терзала всех пользователей, а единственным лечением был аппаратный сброс.

дет такой же, как в примере 19.6, где мы заменили `await asyncio.sleep(3)` на `time.sleep(3)`: никакого вращения. Поток управления переходит от `supervisor` к `slow` и затем к `is_prime`. Когда `is_prime` возвращается, то же самое делает и `slow`, и `supervisor` возобновляет работу и отменяет задачу `spinner`, не дав ей выполниться даже один раз. Выглядит это так, будто программа зависла на 3 с, а затем выдала ответ.

Здоровый сон с помощью sleep(0)

Один из способов не дать индикатору умереть – переписать `is_prime` в виде сопрограммы, которая периодически вызывает `asyncio.sleep(0)` в выражении `await`, чтобы уступить управление циклу событий.

Пример 19.11. spinner_async_nap.py: `is_prime` – теперь сопрограмма

```
async def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    root = math.isqrt(n)
    for i in range(3, root + 1, 2):
        if n % i == 0:
            return False
        if i % 100_000 == 1:
            await asyncio.sleep(0) ❶
    return True
```

❶ Спать после каждых 50 000 итераций (потому что шаг в функции `range` равен 2).

В проблеме 284 (*https://github.com/python/asyncio/issues/284*) в репозитории `asyncio` имеется содержательное обсуждение использования `asyncio.sleep(0)`.

Однако имейте в виду, что это замедляет работу `is_prime` и – что еще важнее – цикл событий, а вместе с ним и всю программу. Когда я вставлял `await asyncio.sleep(0)` через каждые 100 000 итераций, индикатор крутился плавно, но программа на моей машине работала 4.9 с, т. е. почти на 50 % дольше, чем при использовании оригинальной функции `primes.is_prime` с тем же аргументом (`5_000_111_000_222_021`).

Использование `await asyncio.sleep(0)` следует рассматривать как временную меру перед рефакторингом асинхронного кода с целью делегирования длительных вычислений другому процессу. Как это сделать с помощью функции `asyncio.loop.run_in_executor` (*https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.loop.run_in_executor*), мы увидим в главе 21. Еще один вариант – организовать очередь задач – мы кратко обсудим в разделе «Распределенные очереди задач» ниже.

До сих пор мы экспериментировали лишь с одним обращением к счетной функции. В следующем разделе представлено конкурентное выполнение нескольких счетных вызовов.

Доморощенный пул процессов

Я написал этот раздел, чтобы продемонстрировать использование нескольких процессов для решения счетных задач, а также типичный паттерн использования очередей для распределения задач и сбора результатов. В главе 20 будет показан более простой способ распределения задач между процессами: класс `ProcessPoolExecutor` из пакета `concurrent.futures`, который внутри себя пользуется очередями.

В этом разделе мы напишем программы, которые проверяют на простоту выборку из 20 чисел от 2 до 9 999 999 999 999 999, т. е. $10^{16} - 1$, больше чем 2^{53}. Выборка содержит большие и малые простые числа, а также составные числа с большими и малыми простыми множителями.

Программа *sequential.py* определяет эталон для сравнения. Вот пример ее прогона:

```
$ python3 sequential.py
                2 P 0.000001s
 142702110479723 P 0.568328s
 299593572317531 P 0.796773s
3333333333333301 P 2.648625s
3333333333333333   0.000007s
3333335652092209   2.672323s
4444444444444423 P 3.052667s
4444444444444444   0.000001s
4444444488888889   3.061083s
5555553133149889   3.451833s
5555555555555503 P 3.556867s
5555555555555555   0.000007s
6666666666666666   0.000001s
6666666666666719 P 3.781064s
6666667141414921   3.778166s
7777777536340681   4.120069s
7777777777777753 P 4.141530s
7777777777777777   0.000007s
9999999999999917 P 4.678164s
9999999999999999   0.000007s
Total time: 40.31
```

Три столбца интерпретируются следующим образом:

- проверяемое число;
- `P`, если число простое, иначе пробел;
- время проверки этого числа на простоту.

В этом примере полное время работы приближенно равно сумме времен каждой проверки, но вычисляется независимо, как показано в примере 19.12.

Пример 19.12. sequential.py: последовательная проверка на простоту для небольшого набора данных

```
#!/usr/bin/env python3

"""
sequential.py: эталон для сравнения последовательного, многопроцессного и
многопоточного кода счетной задачи.
"""
from time import perf_counter
from typing import NamedTuple

from primes import is_prime, NUMBERS

class Result(NamedTuple): ❶
    prime: bool
    elapsed: float

def check(n: int) -> Result: ❷
    t0 = perf_counter()
    prime = is_prime(n)
    return Result(prime, perf_counter() - t0)

def main() -> None:
    print(f'Checking {len(NUMBERS)} numbers sequentially:')
    t0 = perf_counter()
    for n in NUMBERS: ❸
        prime, elapsed = check(n)
        label = 'P' if prime else ' '
        print(f'{n:16} {label} {elapsed:9.6f}s')

    elapsed = perf_counter() - t0 ❹
    print(f'Total time: {elapsed:.2f}s')

if __name__ == '__main__':
    main()
```

❶ Функция `check` (см. следующий маркер) возвращает кортеж `Result`, содержащий булево значение, возвращенное вызовом `is_prime`, и время его вычисления.
❷ `check(n)` вызывает `is_prime(n)` и вычисляет время, потребовавшееся для возврата `Result`.
❸ Для каждого числа из выборки вызываем `check` и отображаем результат.
❹ Вычислить и показать полное время работы.

Решение на основе процессов

В следующем примере, *procs.py*, показано использование нескольких процессов для распределения проверок на простоту между процессорными ядрами. А это полученные результаты:

```
$ python3 procs.py
Checking 20 numbers with 12 processes:
               2 P 0.000002s
3333333333333333   0.000021s
4444444444444444   0.000002s
```

```
5555555555555555   0.000018s
6666666666666666   0.000002s
 142702110479723 P 1.350982s
7777777777777777   0.000009s
 299593572317531 P 1.981411s
9999999999999999   0.000008s
3333333333333301 P 6.328173s
3333335652092209   6.419249s
4444444488888889   7.051267s
4444444444444423 P 7.122004s
5555553133149889   7.412735s
5555555555555503 P 7.603327s
6666666666666719 P 7.934670s
6666667141414921   8.017599s
7777777536340681   8.339623s
7777777777777753 P 8.388859s
9999999999999917 P 8.117313s
20 checks in 9.58s
```

Последняя строка показывает, что *procs.py* работала в 4.2 раза быстрее, чем *sequential.py*.

Интерпретация времени работы

Отметим, что время работы в третьем столбце относится только к проверке данного конкретного числа. Например, на вычисление `is_prime(7777777777777753)` ушло почти 8.4 с. А в это время другие процессы параллельно проверяли другие числа.

Всего нужно было проверить 20 чисел. Я написал *procs.py*, так что количество рабочих процессов равно числу процессорных ядер, возвращенному функцией `multiprocessing.cpu_count()`.

Общее время в этом случае гораздо меньше суммы времен отдельных проверок. С контекстным переключением процессов и межпроцессным взаимодействием сопряжены некоторые накладные расходы, поэтому в итоге мы получили ускорение только в 4,2 раза по сравнению с последовательной версией. Это хорошо, но немного разочаровывает, учитывая, что программа запустила 12 процессов, чтобы задействовать все имеющиеся ядра.

Функция `multiprocessing.cpu_count()` возвращает 2 на моем MacBook Pro. На самом деле это 6-ядерный Core-i7, но ОС сообщает о 12 процессорах, потому что включен режим гипертрединга – технология Intel, позволяющая выполнять 2 потока на одном ядре. Однако результаты гипертрединга лучше, если один из потоков, занимающих ядро, работает не так интенсивно, как второй, – скажем, первый ждет данных после непопадания в кеш, а в это время второй производит арифметические операции. Как бы то ни было, бесплатных завтраков не бывает: производительность этого ноутбука для счетной задачи, потребляющей мало памяти, например проверки числа на простоту, соответствует 6-ядерной машине.

Код проверки на простоту для многоядерной машины

Когда мы делегируем вычисления процессам или потокам, программа не вызывает рабочую функцию напрямую, поэтому мы не можем просто так получить возвращенное ей значение. Вместо этого исполнитель управляется библиотекой, и она в конечном итоге возвращает результат, который нужно где-то сохранить. Координация исполнителей и сбор результатов – типичные примеры применения очередей в конкурентном программировании, а также в распределенных системах.

Значительная часть нового кода в *procs.py* связана с организацией и использованием очередей. Начало файла показано в примере 19.13.

Класс `SimpleQueue` был добавлен в пакет `multiprocessing` в версии Python 3.9. Если вы пользуетесь более ранней версией, то можете заменить `SimpleQueue` на `Queue` в примере 19.13.

Пример 19.13. procs.py: многопроцессная проверка на простоту; импорт, типы и функции

```
import sys
from time import perf_counter
from typing import NamedTuple
from multiprocessing import Process, SimpleQueue, cpu_count ❶
from multiprocessing import queues ❷

from primes import is_prime, NUMBERS

class PrimeResult(NamedTuple): ❸
    n: int
    prime: bool
    elapsed: float

JobQueue = queues.SimpleQueue[int] ❹
ResultQueue = queues.SimpleQueue[PrimeResult] ❺

def check(n: int) -> PrimeResult: ❻
    t0 = perf_counter()
    res = is_prime(n)
    return PrimeResult(n, res, perf_counter() - t0)

def worker(jobs: JobQueue, results: ResultQueue) -> None: ❼
    while n := jobs.get(): ❽
        results.put(check(n)) ❾
    results.put(PrimeResult(0, False, 0.0)) ❿

def start_jobs(
    procs: int, jobs: JobQueue, results: ResultQueue ⓫
) -> None:
    for n in NUMBERS:
        jobs.put(n) ⓬
    for _ in range(procs):
        proc = Process(target=worker, args=(jobs, results)) ⓭
        proc.start() ⓮
        jobs.put(0) ⓯
```

❶ Стремясь эмулировать `threading`, пакет `multiprocessing` предоставляет `multiprocessing.SimpleQueue`, но это метод, связанный с предопределенным экземпляром низкоуровневого класса `BaseContext`. Мы должны вызвать этот `SimpleQueue`, чтобы построить очередь, но не можем использовать его в аннотациях типов.

❷ В модуле `multiprocessing.queues` есть класс `SimpleQueue`, который нужен нам в аннотациях типов.

❸ `PrimeResult` включает число, проверяемое на простоту. Хранение `n` вместе с другими полями результата впоследствии упростит отображение результатов.

❹ Это псевдоним типа `SimpleQueue`, которым функция `main` (пример 19.14) будет пользоваться для отправки чисел процессам-исполнителям.

❺ Псевдоним второго типа `SimpleQueue`, который будет использован для сбора результатов в `main`. В очереди будут храниться кортежи, состоящие из проверяемого на простоту числа и кортежа `Result`.

❻ Это похоже на *sequential.py*.

❼ `worker` получает очередь подлежащих проверке чисел и другую очередь, в которую будет помещать результаты.

❽ В этой программе я использую число `0` как *отравленную таблетку*: сигнал исполнителю о необходимости завершиться. Если `n` не равно `0`, то цикл продолжается[1].

❾ Инициировать проверку на простоту и поместить `PrimeResult` в очередь.

❿ Отправить `PrimeResult(0, False, 0.0)` обратно, чтобы главный цикл знал, что этот исполнитель работу закончил.

⓫ `procs` – количество процессов, которые будут параллельно проверять числа.

⓬ Поместить подлежащие проверке числа в очередь `jobs`.

⓭ Создать дочерние процессы для всех исполнителей. Каждый дочерний процесс будет исполнять цикл в собственном экземпляре функции `worker`, пока не извлечет `0` из очереди `jobs`.

⓮ Запустить все дочерние процессы.

⓯ Поместить в очередь по одному значению `0` для каждого процесса, чтобы завершить их.

Циклы, сигнальные маркеры и отравленные таблетки

Функция `worker` в примере 19.13 следует общему паттерну конкурентного программирования: выбирать элементы из очереди в бесконечном цикле и обрабатывать каждый в функции, которая выполняет фактическую работу. Цикл завершается, когда из очереди извлечен сигнальный маркер. В этом паттерне сигнальный маркер, останавливающий исполнителя, часто называют «отравленной таблеткой».

[1] В этом примере `0` – удобный сигнальный маркер. Также для этой цели часто используется значение `None`. Но `0` позволяет упростить аннотацию типа в `PrimeResult` и код исполнителя.

В роли сигнального маркера нередко выступает значение `None`, но это не годится, если оно может встречаться в потоке данных. Вызов `object()` – распространенный способ получит уникальное значение, которое может служить сигнальным маркером. Однако этот способ не работает, если процессов несколько, потому что объекты Python необходимо сериализовать для передачи в другой процесс, а когда мы применяем сначала `pickle.dump`, а затем `pickle.load` к экземпляру `object`, десериализованный экземпляр оказывается отличен от оригинала. Хорошей альтернативой `None` может служить встроенный объект `Ellipsis` (или ...), который не теряет своей идентичности в процессе сериализации[1].

В стандартной библиотеке Python в качестве сигнальных маркеров используется много разных значений (*https://mail.python.org/archives/list/python-dev@python.org/message/JBYXQH3NV3YBF7P2HLHB5CD6V3GVTY55/*). В документе PEP 661 «Sentinel Values» (*https://peps.python.org/pep-0661/*) предложен стандартный тип сигнального маркера. Но по состоянию на сентябрь 2021 года он еще не был утвержден.

Теперь рассмотрим функцию `main` в файле *procs.py*.

Пример 19.14. procs.py: многопроцессная проверка на простоту: функция `main`

```
def main() -> None:
    if len(sys.argv) < 2: ❶
        procs = cpu_count()
    else:
        procs = int(sys.argv[1])

    print(f'Checking {len(NUMBERS)} numbers with {procs} processes:')
    t0 = perf_counter()
    jobs: JobQueue = SimpleQueue() ❷
    results: ResultQueue = SimpleQueue()
    start_jobs(procs, jobs, results) ❸
    checked = report(procs, results) ❹
    elapsed = perf_counter() - t0
    print(f'{checked} checks in {elapsed:.2f}s') ❺

def report(procs: int, results: ResultQueue) -> int: ❻
    checked = 0
    procs_done = 0
    while procs_done < procs: ❼
        n, prime, elapsed = results.get() ❽
        if n == 0: ❾
            procs_done += 1
        else:
            checked += 1 ❿
            label = 'P' if prime else ' '
            print(f'{n:16} {label} {elapsed:9.6f}s')
    return checked

if __name__ == '__main__':
    main()
```

[1] Пережить сериализацию, не потеряв своей идентичности, – неплохая цель в жизни.

❶ Если аргументы в командной строке не заданы, то положить количество процессов равным количеству процессорных ядер, в противном случае создать столько процессов, сколько указано в первом аргументе.
❷ `jobs` и `results` – очереди, описанные в примере 19.13.
❸ Запустить `proc` процессов, которые будут выбирать данные из очереди `jobs` и посещать результаты в `results`.
❹ Извлечь и отобразить результаты; функция `report` определена в точке ❻.
❺ Показать количество проверенных чисел и общее затраченное время.
❻ В качестве аргументов передаются количество процессов `procs` и очередь для хранения результатов.
❼ Цикл продолжается, пока не завершатся все дочерние процессы.
❽ Получить один `PrimeResult`. Вызов метода очереди `.get()` блокирует выполнение до тех пор, пока в очереди не появится элемент. Можно также сделать этот вызов неблокирующим или задать тайм-аут. Детали см. в документации по `SimpleQueue.get` (*https://docs.python.org/3/library/queue.html#queue.SimpleQueue.get*).
❾ Если `n` равно 0, то один процесс завершился; увеличить счетчик `procs_done`.
❿ В противном случае увеличить счетчик `checked` (в котором хранится количество проверенных чисел) и отобразить результаты.

Результаты поступают не в том порядке, в каком подавались задания. Поэтому я и включил `n` в кортеж `PrimeResult`. Иначе я не смог бы сопоставить результат с проверенным числом.

Если главный процесс завершится раньше, чем закончат работу все дочерние, то можно будет увидеть странные трассы вызовов в исключениях `FileNotFoundError`, вызванные внутренней блокировкой в пакете `multiprocessing`. Отлаживать конкурентный код вообще трудно, а отлаживать `multiprocessing` еще труднее, поскольку за фасадом «потокоподобия» скрывается немалая сложность. К счастью, класс `ProcessPoolExecutor`, с которым мы познакомимся в главе 20, проще использовать, и он надежнее.

Спасибо читателю Майклу Альберту, который заметил, что в коде примера 19.14, опубликованном в предварительном варианте книги, было *состояние гонки* (*https://en.wikipedia.org/wiki/Race_condition*). Это ошибка, которая может проявиться или не проявиться в зависимости от порядка действий, выполняемых конкурентными единицами выполнения. Если «A» происходит раньше «B», то все хорошо, но если первым происходит «B», значит, что-то пошло не так. Это и есть гонка.

Для особо любопытных я оставил дельту, показывающую, в чем состояла ошибка и как я ее исправил, см. *https://github.com/fluentpython/example-code-2e/commit/2c1230579db99738a5e5e6802063bda585f6476d*, но отмечу, что впоследствии я переработал пример, переместив части `main` в функции `start_jobs` и `report`. В том же каталоге имеется файл *README.md*, где объясняется проблема и ее решение.

Эксперименты с большим и меньшим числом процессов

Можете попробовать выполнить *procs.py*, указав в командной строке количество рабочих процессов. Например, следующая команда

```
$ python3 procs.py 2
```

запускает два рабочих процесса и работает почти в два раза быстрее, чем *sequential.py*, при условии что компьютер оснащен хотя бы двумя ядрами и не слишком занят выполнением других программ.

Я выполнил *procs.py* с числом процессов от 1 до 20 по 12 раз для каждого значения, всего 240 прогонов. Затем вычислил медианное время всех прогонов с одним и тем же числом процессов и построил график на рис. 19.2.

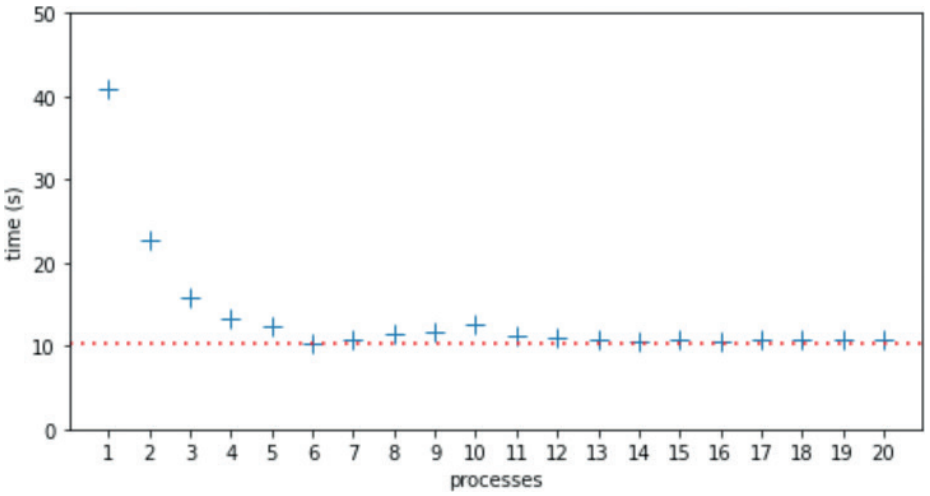


Рис. 19.2. Медианное время работы для каждого числа процессов от 1 до 20. Наибольшее медианное время составило 40,81 с, когда был всего 1 процесс. Наименьшее время равно 10,39 с при 6 процессах, оно обозначено пунктирной линией

На этом 6-ядерном ноутбуке наименьшее медианное время было достигнуто при 6 процессах: 10.39 с, обозначенное пунктирной линией на рис. 19.2. Я ожидал, что время работы начнет расти после 6 процессов из-за конкуренции за процессоры, и оно действительно достигло локального максимума при 10 процессах. Но вот чего я не ожидал и что не могу объяснить – почему производительность улучшилась при 11 процессах и оставалась почти постоянной при числе процессов от 13 до 20, при этом медианное время лишь немного превышает минимум, достигнутый для 6 процессов.

Не решение на основе потоков

Я также написал скрипт *threads.py*, вариант *procs.py*, в котором вместо `multiprocessing` используется `threading`. Код очень похож – так обычно и бывает, когда в простых примерах переходишь от одного из этих API к другому[1]. Из-за GIL и счетного характера функции `is_prime` многопоточная версия медленнее, чем последовательный код в примере 19.12, и замедляется по мере увеличения числа потоков, т. к. растет конкуренция за процессоры и стоимость контекстного переключения. Чтобы переключиться на другой поток, ОС должна сохранить регистры процессора и изменить счетчик программы и указатель стека, что влечет за собой дорогостоящие побочные эффекты,

[1] См. каталог *19-concurrency/primes/threads.py* в репозитории кода.

например недействительность процессорных кешей и, возможно, выгрузку страниц памяти[1].

В следующих двух главах мы будем еще говорить о конкурентном программировании на Python с использованием высокоуровневой библиотеки *concurrent.futures* для управления потоками и процессами (глава 20) и библиотеки *asyncio* для асинхронного программирования (глава 21).

А в оставшихся разделах этой главы постараемся ответить на вопрос:

> С учетом всех описанных выше ограничений как Python удается процветать в мире многоядерных компьютеров?

Python в многоядерном мире

Рассмотрим следующую цитату из хорошо известной статьи Герба Саттера «The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software» (Бесплатные завтраки кончились: фундаментальный поворот к параллелизму в программном обеспечении) (*http://www.gotw.ca/publications/concurrency-ddj.htm*):

> Основные производители и архитектуры процессоров, от Intel и AMD до Sparc и PowerPC, практически исчерпали большинство традиционных подходов к увеличению производительности процессоров. От поднятия тактовой частоты и прямолинейного увеличения пропускной способности команд массово поворачиваются в сторону гипертрединга и многоядерных архитектур. Март 2005 года. (Доступно в сети.)

Саттер называет «бесплатным завтраком» стремление заставить программы работать быстрее без дополнительных усилий со стороны разработчика, потому что процессоры с каждым годом исполняли последовательный код все быстрее и быстрее. Но начиная с 2004 года это уже не так: тактовая частота и оптимизация выполнения достигли плато, и теперь всякое сколько-нибудь существенное повышение производительности возможно только благодаря задействованию нескольких ядер или гипертрединга, а эти преимущества доступны лишь коду, специально написанному для конкурентного выполнения.

История Python началась в начале 1990-х годов, когда производительность выполнения последовательного кода все еще росла по экспоненте. Тогда о многоядерных процессорах говорили только в контексте суперкомпьютеров. В то время решение завести GIL представлялось очевидным. Благодаря GIL интерпретатор работает быстрее на одном ядре, а его реализация упрощается[2]. Кроме того, GIL упрощает написание простых расширений с помощью Python/C API.

---

[1] Дополнительные сведения см. в статье Википедии «Context switch» (*https://en.wikipedia.org/wiki/Context_switch*).

[2] Вероятно, по тем же причинам Юкихиро Мацумото, создатель языка Ruby, тоже решил использовать GIL в своем интерпретаторе.

Я написал «простые расширения», потому что расширение вообще-то не обязано иметь хоть какое-то отношение к GIL. Функция, написанная на C или на Fortran, может работать в сотни раз быстрее, чем эквивалентная ей на Python[1]. Поэтому дополнительная сложность, связанная с освобождением GIL, чтобы можно было воспользоваться всеми ядрами многоядерного процессора, во многих случаях и не нужна. Так что мы можем поблагодарить GIL за многочисленные расширения, доступные для Python, а это, конечно, одна из причин, по которым язык так популярен сегодня.

Несмотря на GIL, Python прекрасно себя чувствует в приложениях, нуждающихся в конкурентном или параллельном выполнении, благодаря библиотекам и программным архитектурам, которые позволяют обойти ограничения CPython.

Теперь обсудим, как Python применяется для системного администрирования, в науке о данных и при разработке серверных приложений в распределенном мире многоядерных компьютеров, существующем в 2021 году.

Системное администрирование

Python широко применяется для управления большими парками серверов, маршрутизаторов, балансировщиков нагрузки и сетевых устройств хранения (NAS). Он также занимает лидирующие позиции в области программно-конфигурируемых сетей (SDN) и этического хакинга. Главные поставщики облачных служб поддерживают Python в своих библиотеках и пособиях, написанных как самими поставщиками, так и многочисленными сложившимися вокруг них сообществами пользователей Python.

В этой области Python-скрипты автоматизируют задачи конфигурирования, выдавая команды, подлежащие выполнению удаленными машинами, поэтому счетные операции встречаются редко. Потоки и сопрограммы прекрасно приспособлены для задач такого рода. В частности, пакет `concurrent.futures`, который мы будем изучать в главе 20, можно использовать для выполнения одних и тех же операций на большом числе удаленных машин одновременно, не сильно увеличивая сложность.

Помимо стандартной библиотеки, существуют популярные основанные на Python проекты для управления кластерами серверов: инструменты типа *Ansible* (*https://www.ansible.com/*) и *Salt*, а также библиотеки типа *Fabric* (*https://www.fabfile.org/*).

Постоянно растет число библиотек для системного администрирования с поддержкой сопрограмм и `asyncio`. В 2016 году группа организации производства компании Facebook писала в отчете (*https://engineering.fb.com/2016/05/27/production-engineering/python-in-production-engineering/*): «Мы все сильнее полагаемся на пакет AsyncIO, появившийся в версии Python 3.4, и наблюдаем огромный рост производительности при переводе кодовой базы с Python 2».

[1] В колледже я должен был в качестве упражнения реализовать на C алгоритм сжатия LZW. Но сначала я написал его на Python, чтобы проверить, правильно ли понял спецификацию. Так вот, версия на C оказалась примерно в 900 раз быстрее.

Наука о данных

Для науки о данных, включая искусственный интеллект, и научных расчетов в Python имеется прекрасная поддержка. В этих предметных областях приложения в основном счетные, но у пользователей Python есть преимущество – обширная экосистема библиотек для численных расчетов, написанных на C, C++, Fortran, Cython и т. д., многие из которых способны задействовать несколько ядер, GPU и (или) распределенные параллельные вычисления в гетерогенных кластерах.

По состоянию на 2021 год экосистема науки о данных в Python включала, в частности, следующие впечатляющие инструменты:

Проект Jupyter (https://jupyter.org/)

Два основанных на браузере интерфейса, Jupyter Notebook и JupyterLab, позволяют пользователям выполнять и документировать аналитический код, который может при необходимости работать на удаленных машинах. Оба являются гибридными приложениями, написанными на Python и JavaScript, поддерживают вычислительные ядра, написанные на других языках, и интегрированы с помощью ZeroMQ, библиотеки организации асинхронных сообщений для распределенных приложений. Название *Jupyter* происходит от Julia, Python и R – первых трех языков, поддержанных системой Notebook. Развитая экосистема, построенная поверх инструментария Jupyter, включает Bokeh (*https://docs.bokeh.org/en/latest/index.html*), мощную интерактивную библиотеку визуализации, которая дает пользователям возможность взаимодействовать с большими наборами данных или непрерывно обновляемыми потоковыми данными – благодаря высокой производительности современных движков JavaScript и браузеров.

TensorFlow и PyTorch

Это две самые популярные библиотеки глубокого обучения, согласно отчету издательского дома O'Reilly за январь 2021 года (*https://www.oreilly.com/radar/where-programming-ops-ai-and-the-cloud-are-headed-in-2021/*) об использовании их образовательных ресурсов в течение 2020 года. Оба проекта написаны на C++ и способны задействовать несколько ядер, GPU и кластеры. Они имеют интерфейсы и к другим языкам, но Python стоит на первом месте и имеет наибольшее число пользователей. TensorFlow была создана в Google и используется там для внутренних целей; PyTorch разработана компанией Facebook.

Dask

Библиотека параллельных вычислений, которая может распределять работу между локальными процессами или кластерами компьютеров, «протестирована на некоторых из крупнейших в мире суперкомпьютеров», как утверждается на домашней странице проекта (*https://dask.org/*). Dask предлагает API, имеющие много общего с NumPy, pandas и scikit-learn – самыми популярными библиотеками в науке о данных и машинном обучении на сегодняшний день. Dask можно использовать из JupyterLab или Jupyter Notebook, она обращается к Bokeh не только для визуализации данных, но и как к интерактивной контрольной панели, на которой отображаются пото-

ки данных и вычислений в различных процессах и компьютерах в режиме, близком к реальному времени. Dask настолько впечатляет, что я рекомендую посмотреть видео, например 15-минутный ролик по адресу *https://www.youtube.com/watch?v=ods97a5Pzw0*, в котором Мэттью Роклин, отвечающий за сопровождение проекта, демонстрирует, как Dask перемалывает данные на 64 ядрах, распределенных между 8 машинами типа EC2 в облаке AWS.

И это лишь несколько примеров, иллюстрирующих, как сообщество науки о данных создает решения, способные использовать лучшее из предлагаемого Python и преодолевать ограничения среды выполнения CPython.

Веб-разработка на стороне сервера и на мобильных устройствах

Python широко используется в веб-приложениях и для создания серверных API, поддерживающих мобильные приложения. Как Google, YouTube, Dropbox, Instagram, Quora и Reddit (в числе прочих) сумели построить на Python серверные приложения, обслуживающие сотни миллионов пользователей в режиме 24×7? Ответ выходит за рамки того, что Python предлагает «из коробки».

Прежде чем обсуждать инструменты для поддержки Python в крупномасштабных приложениях, я обязан процитировать увещевание из издания «Technology Radar» компании Thoughtworks:

> **Зависть к высокой производительности и масштабу веба**
>
> Мы видели, как многие команды попадали впросак, т. к. выбирали сложные инструменты, каркасы или архитектуры, потому что «в будущем, возможно, придется масштабироваться». Такие компании, как Twitter и Netflix, вынуждены поддерживать экстремальные нагрузки, поэтому не могут обойтись без подобных архитектур, но они располагают персоналом высочайшей квалификации, способным справиться с их сложностью. В большинстве ситуаций такие инженерные подвиги ни к чему; команде следует отказаться от зависти к масштабу веба в пользу более простых решений, позволяющих решить задачу[1].

В *масштабе веба* ключом является архитектура, допускающая горизонтальное масштабирование. На этом уровне все системы распределенные, и вряд ли найдется единственный язык программирования, который лучше всего отвечает потребностям каждой части решения.

Распределенные системы – область академических исследований, но, к счастью, есть доступные книги, написанные практиками и основанные на фундаментальных исследованиях и практическом опыте. Один из таких практиков – Мартин Клеппманн, автор книги «Designing Data-Intensive Applications» (O'Reilly).

Взгляните на рис. 19.3, первую из многочисленных архитектурных диаграмм в книге Клеппманна. Приведу несколько компонентов, которые встре-

[1] Источник: «Консультативный совет по технологиям Thoughtworks», *Technology Radar* ноябрь 2015 (*https://www.thoughtworks.com/radar/techniques/high-performance-envy-web-scale-envy*).

чались мне в использующих Python организациях, где я работал или о которых я знаю не понаслышке:

- кеши приложений[1]: *memcached*, *Redis*, *Varnish*;
- реляционные базы данных: *PostgreSQL*, *MySQL*;
- документные базы данных: *Apache CouchDB*, *MongoDB*;
- полнотекстовые индексы: *Elasticsearch*, *Apache Solr*;
- очереди сообщений: *RabbitMQ*, *Redis*.

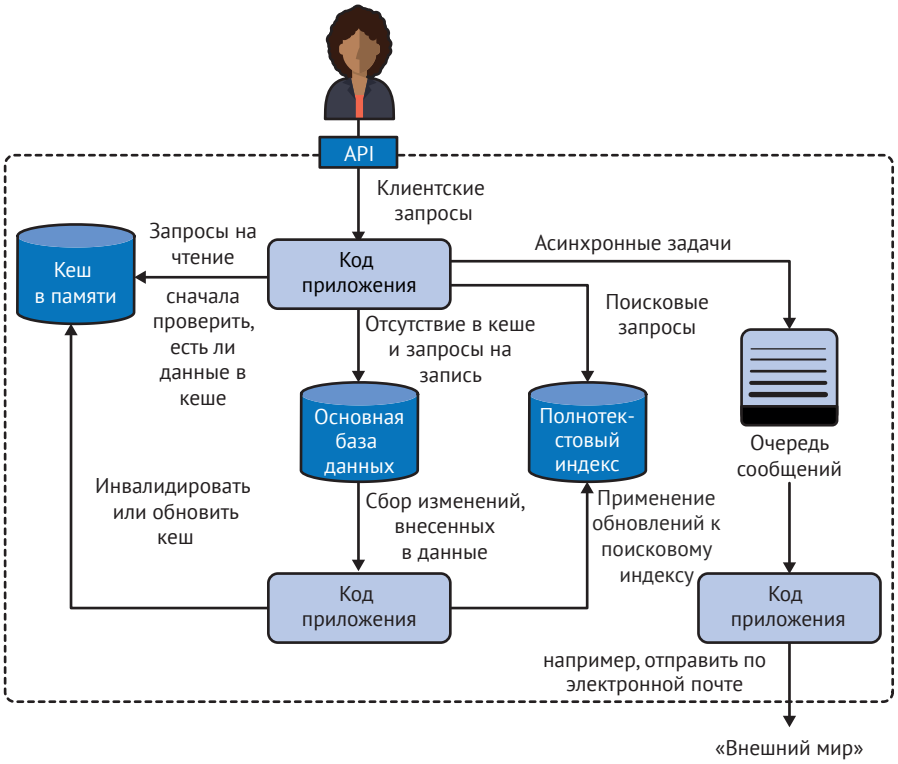


Рис. 19.3. Одна из возможных архитектур для системы, содержащей несколько компонентов[2]

В каждой из этих категорий существуют и другие продукты с открытым исходным кодом производственного уровня. Крупные облачные поставщики также предлагают собственные альтернативы.

Диаграмма Клеппманна является общей и от языка не зависит – как и вся его книга. В серверных приложениях на Python часто развертываются два конкретных компонента:

---

[1] Не путайте кеши приложений (используемые напрямую кодом вашего приложения) с HTTP-кешами, которые должны бы находиться на самом верху рис. 19.3 и предназначены для обслуживания запросов к статическим ресурсам: изображениям, CSS- и JS-файлам и т. д. Сети доставки контента (CDN) предлагают еще один тип HTTP-кеша, который развертывается в центрах обработки данных, ближайших к конечным пользователям приложения.

[2] Диаграмма основана на рис. 1.1 из книги Martin Kleppmann «Designing Data-Intensive Applications» (O'Reilly).

- сервер приложений для распределения нагрузки между несколькими экземплярами приложения. Этот сервер должен находиться в верхней части рис. 19.3 и обслуживать клиентские запросы, до того как они достигнут кода приложения;
- очередь задач, построенная на базе очереди сообщений в правой части рис. 19.3. Она предоставляет высокоуровневый простой в использовании API для распределения задач между процессами, работающими на других машинах.

В следующих двух разделах рассматриваются компоненты этого типа, которые рекомендуются для развертывания в серверных Python-приложениях.

WSGI-серверы приложений

WSGI – шлюзовой интерфейс веб-серверов (*https://peps.python.org/pep-3333/*) – это стандартный API Python-каркаса или приложения для получения запросов от HTTP-сервера и отправки ему ответов[1]. WSGI-серверы приложений управляют одним или несколькими процессами, исполняющими приложение, обеспечивая максимальное использование доступных процессоров.

На рис. 19.4 показано типичное развертывание WSGI.

Если бы мы захотели объединить обе диаграммы, то содержимое обведенного штриховой линией прямоугольника на рис. 19.4 следовало бы поместить вместо блока «Код приложения» в верхней части рис. 19.3.

Из самых известных серверов приложений для веб-проектов на Python упомянем следующие:

- *mod_wsgi* (*https://modwsgi.readthedocs.io/en/master/*);
- *uWSGI* (*https://uwsgi-docs.readthedocs.io/en/latest/*)[2];
- *Gunicorn* (*https://gunicorn.org/*);
- *NGINX Unit* (*https://unit.nginx.org/*).

Для пользователей HTTP-сервера Apache лучшим вариантом является *mod_wsgi*. Он такой же старый, как сам стандарт WSGI, но активно сопровождается, а теперь еще и предлагает средство запуска из командной строки `mod_wsgi-express`, которое упрощает конфигурирование и больше подходит для использования в Docker-контейнерах.

---

[1] Одни докладчики произносят WSGI как акроним, по буквам, другие – как одно слово, «висги».

[2] *uWSGI* пишется со строчной буквой «u», но произносится она как греческая буква «μ», так что название в целом произносится как «микровисги».

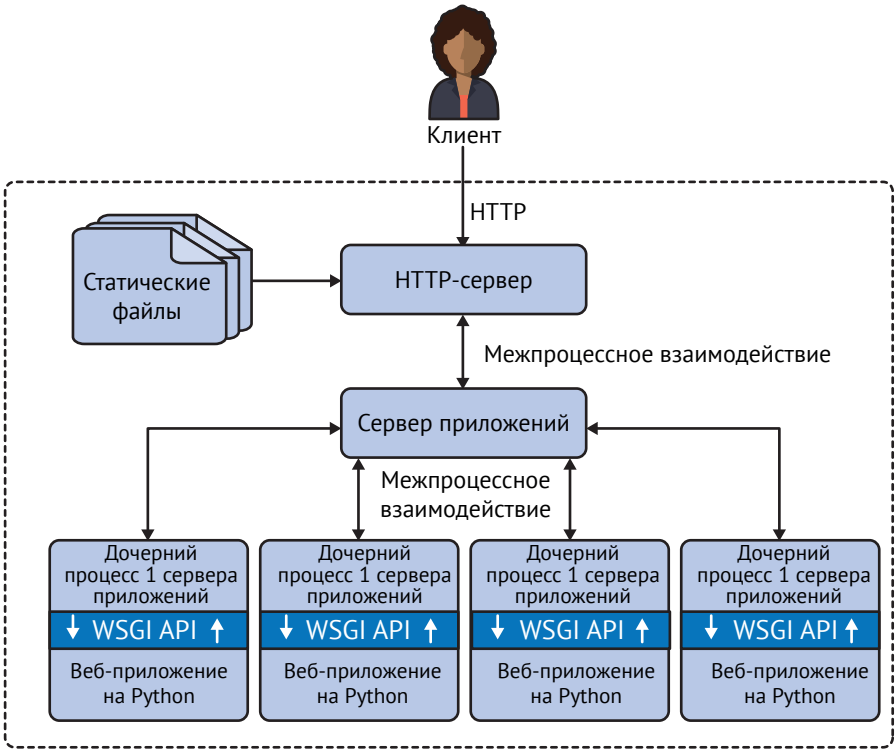


Рис. 19.4. Клиенты подключаются к HTTP-серверу, который доставляет статические файлы и передает другие запросы серверу приложений, а тот запускает дочерние процессы для выполнения кода приложения, задействуя несколько процессорных ядер. WSGI API – связующий слой между сервером приложений и кодом Python-приложения

uWSGI и *Gunicorn* чаще всего выбираются в недавних проектах, о которых мне известно. Оба нередко применяются в сочетании с HTTP-сервером *NGINX*. *uWSGI* предлагает дополнительную функциональность, включая кеш приложения, очередь задач, планировщик периодических задач типа cron и много всего другого. Но правильно сконфигурировать *uWSGI* гораздо труднее, чем *Gunicorn*[1].

NGINX Unit – новый продукт, выпущенный в 2018 году авторами широко известного HTTP-сервера и обратного прокси-сервера *NGINX*.

mod_wsgi и *Gunicorn* поддерживают только веб-приложения на Python, тогда как *uWSGI* и *NGINX Unit* работают и с другими языками. Дополнительные сведения можно почерпнуть из соответствующей документации.

[1] Инженеры из компании Bloomberg Петер Шперль и Бен Грин написали руководство «Configuring uWSGI for Production Deployment» (*https://www.bloomberg.com/tosv2.html?vid=&uuid=f46c5dc1-df27-11ec-ba90-464c446b4e6d&url=L2NvbXBhbnkvc3Rvcmllcy9jb25maWd1cmluZy11d3NnaS1wcm9kdWN0a W9uLWRlcGxveW1lbnQv*), в котором объясняется, сколько параметров, подразумеваемых по умолчанию в uWSGI, не годятся для многих типичных сценариев развертывания. Шперль конспективно представил эти рекомендации на конференции EuroPython 2019 (*https://www.youtube.com/watch?v=p6R1h2Nn468*). Горячо рекомендую всем пользователям uWSGI.

Главное: все эти серверы приложений потенциально могут задействовать все процессорные ядра сервера, создавая несколько процессов Python для выполнения традиционных веб-приложений, написанных в старом добром последовательном стиле, например *Django*, *Flask*, *Pyramid* и т. д. Это объясняет, почему веб-разработчик на Python может зарабатывать на хлеб, не изучая модули `threading`, `multiprocessing` или *asyncio*: серверы приложений прозрачно управляют конкурентностью.

ASGI – шлюзовой интерфейс асинхронных серверов

WSGI – синхронный API. Он не поддерживает сопрограмм с `async`/`await` – самый эффективный способ реализации веб-советов или долгого HTTP-опроса на Python. Спецификация ASGI, пришедшая на смену WSGI, спроектирована для асинхронных веб-каркасов на Python, например *aiohttp*, *Sanic*, *FastAPI* и т. д., а равно *Django* и *Flask*, в которые постепенно добавляется асинхронная функциональность.

Теперь обратимся к еще одному способу обхода GIL для достижения более высокой производительности в серверных Python-приложениях.

Распределенные очереди задач

Когда сервер приложений доставляет запрос одному из процессов Python, исполняющих наш код, наше приложение должно ответить быстро: мы хотим, чтобы процесс как можно скорее стал доступен для обслуживания следующего запроса. Однако некоторые запросы требуют выполнения достаточно длительных действий, например отправки по электронной почте или генерирования PDF-документа. Эту проблему и призваны решить распределенные очереди задач.

Наиболее известные очереди задач с открытым исходным кодом, имеющие Python API, – *Celery* (*https://docs.celeryq.dev/en/stable/getting-started/introduction.html) и RQ (https://python-rq.org/*). Облачные поставщики также предлагают собственные очереди задач.

Эти продукты обертывают очередь сообщений и предоставляют высокоуровневый API для делегирования задач исполнителям, возможно, работающим на разных машинах.

В контексте очередей задач вместо традиционной терминологии «клиент» и «сервер» используются слова «производитель» и «потребитель». Например, обработчик представлений в *Django производит* задания-запросы, которые помещаются в очередь для *потребления* одним или несколькими процессами генерирования PDF.

В следующей цитате из *Celery* FAQ (*https://docs.celeryq.dev/en/stable/faq.html#what-kinds-of-things-should-i-use-celery-for*) перечисляются некоторые типичные применения.

- Выполнение чего-то в фоновом режиме. Например, как можно скорее завершить обработку веб-запроса, а затем инкрементно обновить страницу пользователя. Это создает у пользователя впечатление высокой производительности и «шустрости», хотя реальная работа может занять некоторое время.
- Выполнение чего-то после завершения веб-запроса.
- Гарантирование полного завершения чего-то, возможно, асинхронно и за несколько попыток.
- Планирование периодических работ.

Помимо решения этих непосредственных задач, очереди задач поддерживают горизонтальную масштабируемость. Связь между производителями и потребителями разорвана: производитель не вызывает потребителя, он только помещает запрос в очередь. Потребители не должны ничего знать о производителях (но запрос может включать информацию о производителе, если требуется подтверждение). Главное, что можно легко добавлять исполнителей для потребления задач по мере увеличения спроса. Именно поэтому *Celery* и *RQ* называют также распределенными очередями задач.

Напомним, что в нашем простом скрипте *procs.py* (пример 19.13) использовалось две очереди: одна для заданий-запросов, другая для сбора результатов. В распределенных архитектурах *Celery* и *RQ* применяется похожий паттерн. Та и другая поддерживают базу данных NoSQL *Redis* (*https://redis.io/*) в качестве очереди сообщений и хранилища результатов. *Celery* поддерживает и другие очереди сообщений, например *RabbitMQ* и *Amazon SQS*, а также иные базы данных для хранения результатов.

На этом мы завершаем введение в конкурентность в Python. В следующих двух главах эта тема будет продолжена, с упором на пакеты `concurrent.futures` и `asyncio` из стандартной библиотеки.

Резюме

После краткого теоретического введения были представлены скрипты анимированного индикатора, реализованные с помощью каждой из трех имеющихся в Python моделей конкурентного программирования:

- потоки, пакет `threading`;
- процессы, пакет `multiprocessing`;
- асинхронные сопрограммы, пакет `asyncio`.

Затем мы изучили реальное влияние GIL, проведя эксперимент: мы изменили написанные ранее примеры, так чтобы они проверяли большие целые числа на простоту, и понаблюдали за поведением. Было графически продемонстрировано, что при использовании `asyncio` счетных операций следует избегать, поскольку они блокируют цикл событий. Многопоточная версия работала хорошо – несмотря на GIL, – потому что Python периодически прерывает потоки, а в примере использовалось всего два потока: один выполнял счетную операцию, а другой управлял анимацией 10 раз в секунду, что совсем немного. Вариант на основе `multiprocessing` вообще обходил GIL, поскольку запускался новый процесс специально для анимации, тогда как главный процесс занимался проверкой на простоту.

В следующем примере, где проверялось несколько простых чисел, была продемонстрирована разница между пакетами `multiprocessing` и `threading` и показано, что только процессы позволяют Python задействовать преимущества многоядерных CPU. Из-за GIL потоки в Python работают хуже последовательного кода при выполнении длительных вычислений.

Вопросы о GIL преобладают в обсуждениях конкурентных и параллельных вычислений на Python, но переоценивать его значимость не следует. Эта тема была поднята в разделе «Python в многоядерном мире». Например, на GIL можно не обращать внимания во многих сценариях системного администрирования на Python. С другой стороны, сообщества разработчиков в области науки о данных и серверных приложений нашли решения, позволяющие обойти GIL и строить решения производственного уровня, адаптированные к их конкретным потребностям. В последних двух разделах были рассмотрены два компонента, часто применяемых для поддержки крупномасштабных серверных Python-приложений: WSGI-серверы приложений и распределенные очереди задач.

Дополнительная литература

Список для дополнительного чтения к этой главе довольно обширен, поэтому я разбил его на части.

Конкурентность с применением потоков и процессов

В библиотеке *concurrent.futures*, рассматриваемой в главе 20, под капотом используются потоки, процессы, блокировки и очереди, но снаружи вы их не увидите; все они скрыты за фасадом высокоуровневых абстракций `ThreadPoolExecutor` и `ProcessPoolExecutor`. Если вам интересно больше узнать о практике конкурентного программирования с помощью этих низкоуровневых объектов, начните со статьи Джима Андерсона «An Intro to Threading in Python» (*https://realpython.com/intro-to-python-threading/*). Дуг Хеллманн включил главу «Concurrency with Processes, Threads, and Coroutines» в свою книгу «The Python 3 Standard Library by Example» (Addison-Wesley) (*https://www.pearson.com/us/higher-education/program/Hellmann-Python-3-Standard-Library-by-Example-The/PGM328871.html)* и поместил ее на свой сайт *(https://pymotw.com/3/concurrency.html*).

Книги Brett Slatkin «Effective Python», 2-е издание (Addison-Wesley), David Beazley «Python Essential Reference», 4-е издание (Addison-Wesley), и Martelli et al. «Python in a Nutshell», 3-е издание (O'Reilly), посвящены Python в целом, и в них подробно рассматриваются пакеты `threading` и `multiprocessing`. В обширной официальной документации по пакету `multiprocessing` имеются полезные советы в разделе «Рекомендации по программированию» (*https://docs.python.org/3/library/multiprocessing.html#programming-guidelines*).

Джесси Ноллер и Ричард Оудкерк, авторы пакета `multiprocessing`, описали его в документе PEP 371 «Addition of the multiprocessing package to the standard library» (*https://peps.python.org/pep-0371/*). Официальная документация по этому пакету представляет собой *rst*-файл размером 93 КБ (*https://docs.python.org/3/library/ multiprocessing.html*) – приблизительно 63 страницы; это одна из самых длинных глав в документации по стандартной библиотеке.

В книге Micha Gorelick, Ian Ozsvald «High Performance Python», 2-е издание (O'Reilly), есть глава о модуле `multiprocessing` с примером, посвященным проверке на простоту, но с помощью стратегий, отличающихся от примененной в нашем скрипте *procs.py*. Для каждого числа они разбивают диапазон возможных множителей – от 2 до `sqrt(n)` – на поддиапазоны и поручают исполнителям исследовать эти поддиапазоны. Такой подход «разделяй и властвуй» характерен для научных приложений с огромными наборами данных и ситуаций, когда рабочие станции (или кластеры) имеют больше процессорных ядер, чем пользователей. В серверной системе, обрабатывающей запросы от многих пользователей, проще и эффективнее поручить каждому процессу провести вычисление от начала до конца, поскольку это снижает накладные расходы на взаимодействие и координацию процессов. Помимо пакета `multiprocessing`, Горелик и Освальд описывают много других способов разработки и развертывания высокопроизводительных приложений в области науки о данных с использованием нескольких ядер, GPU, кластеров, профилировщиков и компиляторов типа Cython и Numba. В последней главе «Полевые заметки» приводится набор полезных коротких примеров, предложенных другими разработчиками, практически занимающимися высокопроизводительными вычислениями на Python.

Книга Matthew Wilkes «Advanced Python Development» (Apress)[1] – одна из тех редких книг, в которой есть короткие примеры для объяснения идей и в то же время строится реалистичное приложение, готовое к эксплуатации: агрегатор данных, похожий на системы мониторинга в DevOps или системы сбора данных для распределенных датчиков в IoT. В двух главах этой книги рассматривается конкурентное программирование с применением пакетов `threading` и `asyncio`.

В книге Jan Palach «Parallel Programming with Python» (Packt, 2014) объясняются основные идеи, стоящие за конкурентностью и параллелизмом, и рассматривается стандартная библиотека Python и библиотека *Celery*.

Глава 2 книги Caleb Hattingh «Using Asyncio in Python» (O'Reilly) называется «Вся правда о потоках»[2]. В ней речь идет о плюсах и минусах многопоточного программирования – с привлечением убедительных цитат из нескольких авторитетных источников, – так что становится понятно, что фундаментальные проблемы потоков не связаны ни с Python, ни с GIL.

Процитирую фрагмент страницы 14 этой книги:

> Следующие аргументы повторяются снова и снова:
>
> - наличие потоков затрудняет рассуждения о программе;
> - многопоточная модель неэффективна для организации масштабной конкурентности (тысячи конкурентных потоков).

Если вы хотите тяжким трудом, но не рискуя потерять работу, добыть знания о том, как трудно рассуждать о потоках и блокировках, попробуйте решить упражнения из книги Allen Downey «The Little Book of Semaphores» (Green Tea Press) (*greenteapress.com/wp/semaphores*). По трудности они варьируются от совсем простых до нерешаемых, но даже простые способствуют просветлению.

---

[1] *Мэттью Уилкс.* Профессиональная разработка на Python. ДМК Пресс, 2021 // *https://dmkpress.com/catalog/computer/programming/python/978-5-97060-930-9/*

[2] Калеб – один из технических рецензентов второго издания этой книги.

GIL

Если вас заинтриговали тайны GIL, вспомните, что из кода на Python управлять ей нельзя, поэтому каноническим справочником является документация по C-API: «Состояние потока и глобальная блокировка интерпретатора» (*https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock*). *FAQ по библиотеке и расширениям Python* отвечает на вопрос: «Можно ли избавиться от глобальной блокировки интерпретатора?» (*https://docs.python.org/3/faq/library.html#can-t-we-get-rid-of-the-global-interpreter-lock*). Стоит также прочитать статьи Гвидо ван Россума и Джесси Ноллера (автора пакета `multiprocessing`): «It isn't Easy to Remove the GIL» (*https://www.artima.com/weblogs/ viewpost.jsp?thread=214235*) и «Python Threads and the Global Interpreter Lock» (*http://jessenoller.com/blog/2009/02/01/python-threads-and-the-global-interpreter-lock*).

В книге Anthony Shaw «CPython Internals» (*https://realpython.com/products/cpython-internals-book/*) (Real Python) объясняется, как реализован интерпретатор CPython 3 на уровне программирования на C. Самая длинная глава в этой книге – «Параллелизм и конкурентность», это глубокое погружение в поддержку потоков и процессов, включая управление GIL из расширений, следующих C/Python API.

Наконец, Дэвид Бизли представил детальное исследование вопроса в презентации «Understanding the Python GIL» (*http://www.dabeaz.com/GIL/*)[1]. На слайде 54 презентации (*http://www.dabeaz.com/python/UnderstandingGIL.pdf*) Бизли отмечает увеличение времени работы конкретного теста производительности после реализации нового алгоритма GIL в версии Python 3.2. Проблема несущественна для реальных рабочих нагрузок, как заметил (*https://bugs.python.org/issue7946# msg223110*) Антуан Питру, реализовавший этот алгоритм, в ответ на сообщение об ошибке, отправленное Бизли; см. проблему Python #7946 (*https://bugs.python.org/issue7946*).

Конкурентность за пределами стандартной библиотеки

В этой книге рассматриваются прежде всего базовые средства языка и части стандартной библиотеки. Хорошим дополнением к ней является книга «Full Stack Python» (*https://www.fullstackpython.com/*): она посвящена экосистеме Python и среди прочего содержит главы «Среды разработки», «Данные», «Веб-разработка» и «DevOps».

Я уже упоминал две книги, где обсуждается конкурентность с использованием стандартной библиотеки Python, но включен также обширный материал по сторонним библиотекам и инструментам: «High Performance Python», 2-е издание (*https://www.oreilly.com/library/view/high-performance-python/9781492055013/*), и «Parallel Programming with Python» (*https://www.packtpub.com/product/parallel-programming-with-python/9781783288397*). В книге Francesco Pierfederici «Distributed Computing with Python» (*https://www.packtpub.com/product/distributed-computing-with-python/9781785889691*) рассматривается стандартная библиотека, а также использование облаков и высокопроизводительных (HPC) кластеров.

Статья Мэттью Роклина «Python, Performance, and GPUs», опубликованная в июне 2019 года (*https://towardsdatascience.com/python-performance-and-gpus-1be860ffd58d*), содержит актуальную информацию об использовании GPU-ускорителей.

[1] Спасибо Лукасу Бруниалти, который прислал мне эту ссылку.

«Instagram в настоящее время является крупнейшим в мире развертыванием веб-каркаса *Django*, целиком написанного на Python». Так начинается статья в блоге «Web Service Efficiency at Instagram with Python» (*https://instagram-engineering.com/web-service-efficiency-at-instagram-with-python-4976d078e366*), написанная Мин Ни – программистом, работающим в Instagram. В статье описываются метрики и инструменты, которые используются в Instagram для оптимизации кодовой базы на Python, а также обнаружения и диагностики снижения производительности в условиях развертывания серверной части «30–50 раз в день».

В книге Harry Percival, Bob Gregory «Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices» (*https://www.oreilly.com/library/view/architecture-patterns-with/9781492052197/*) (O'Reilly) представлены архитектурные паттерны для серверных Python-приложений. Авторы также выложили книгу в свободный доступ на сайте *cosmicpython.com*.

Существуют две элегантные и очень простые для использования библиотеки распараллеливания задач между процессами: `lelo` (*https://pypi.python.org/pypi/lelo*) Жоао С. О. Буэно и `python-parallelize` (*https://github.com/npryce/python-parallelize*) Ната Прайса. В пакете `lelo` определен декоратор `@parallel`; если применить его к любой функции, то она, как по волшебству, становится неблокирующей, поскольку выполняется в отдельном процессе. Пакет Ната Прайса `python-parallelize` предоставляет генератор `parallelize`, который можно использовать для выполнения цикла `for` на нескольких процессорах. В основе обоих пакетов лежит библиотека `multiprocessing`.

Разработчик ядра Python Эрик Шоу поддерживает вики Multicore Python (*https://github.com/ericsnowcurrently/multi-core-python/wiki*), где рассказывает о своих и сторонних усилиях улучшить поддержку параллельной работы в Python. Сноу – автор документа PEP 554 «Multiple Interpreters in the Stdlib» (*https://peps.python.org/pep-0554/*). Если он будет одобрен и реализован, то станет основой для будущих усовершенствований, которые в конечном итоге, возможно, позволят Python задействовать несколько ядер без накладных расходов, свойственных *многопроцессной обработке*. Одно из основных препятствий – сложное взаимодействие между несколькими активными дочерними интерпретаторами и расширениями, предполагающими, что интерпретатор всего один.

Марк Шеннон – тоже отвечающий за сопровождение Python – создал полезную таблицу (*https://gist.github.com/markshannon/79cace3656b40e21b7021504daee950c*), в которой сравнивает модели конкурентности в Python, упоминавшиеся в обсуждении дочерних интерпретаторов в списке рассылки python-dev (*https://mail.python.org/archives/list/python-dev@python.org/message/YOOQZCFOKEPQ24YHWWLQSJ3RCXFMS7D7/*), в котором принимали участие он сам, Эрик Сноу и другие разработчики. В таблице Шеннона есть столбец «Идеальное CSP», относящийся к теоретической модели взаимодействующих последовательных процессов (*https://en.wikipedia.org/wiki/Communicating_sequential_processes*), предложенной Тони Хоаром в 1978 году. Язык Go также допускает разделяемые объекты, нарушая фундаментальное ограничение CSP: единицы выполнения должны взаимодействовать только с помощью передачи сообщений по каналам.

Stackless Python (*https://github.com/stackless-dev/stackless/wiki*) (или просто *Stackless*) – клон CPython, в котором реализованы микропотоки, т. е. облегченные потоки, управляемые на уровне приложения, а не ОС. Массивно многополь-

зовательская онлайновая игра *EVE Online* (*https://www.eveonline.com/*) построена на базе *Stackless*, а инженеры, нанятые компанией-разработчиком CCP (*https://www.ccpgames.com/*), некоторое время сопровождали (*https://stackless.readthedocs.io/en/3.6-slp/stackless-python.html#history*) *Stackless*. Некоторые особенности *Stackless* были заново реализованы в интерпретаторе *Pypy* (*https://doc.pypy.org/en/latest/stackless.html*) и в пакете *greenlet* (*https://greenlet.readthedocs.io/en/latest/*), лежащем в основе сетевой библиотеки *gevent* (*http://www.gevent.org/*), которая, в свою очередь, является фундаментом сервера приложений *Gunicorn* (*https://gunicorn.org/*).

Модель акторов в конкурентном программировании лежит в основе очень хорошо масштабируемых языков Erlang и Elixir, а также каркаса Akka для Scala и Java. Если вы хотите попробовать модель акторов в Python, обратите внимание на библиотеки *Thespian* (*http://thespianpy.com/doc/*) и *Pykka* (*https://pykka.readthedocs.io/en/latest/*).

В остальных моих рекомендациях Python почти не упоминается, но они все равно могут представлять интерес для читателей, заинтересовавшихся темой этой главы.

Конкурентность и масштабируемость за пределами Python

Книга Alvaro Videla, Jason J. W. Williams «RabbitMQ in Action» (Manning) (*https://www.manning.com/books/rabbitmq-in-action*) – прекрасно написанное введение в систему *RabbitMQ* и стандарт Advanced Message Queuing Protocol (AMQP), с примерами на Python, PHP и Ruby. Вне зависимости от того, что еще есть в вашем техническом загашнике, и даже если вы планируете использовать *Celery*, в которой *RabbitMQ* скрыта под капотом, я рекомендую прочитать эту книгу, поскольку в ней рассматриваются идеи, мотивация и паттерны распределенных очередей сообщений, а также эксплуатация и настройка *RabbitMQ* в крупномасштабных проектах.

Я много узнал из книги Paul Butcher «Seven Concurrency Models in Seven Weeks[1]» (*https://pragprog.com/titles/pb7con/seven-concurrency-models-in-seven-weeks/*) (Pragmatic Bookshelf) с красноречивым подзаголовком «Когда распутываются нити». В главе 1 представлены основные идеи и проблемы программирования с потоками и блокировками в Java[2]. Остальные шесть глав посвящены тому, что автор считает наилучшими альтернативами средствам конкурентного и параллельного программирования, поддерживаемым различными языками, инструментами и библиотеками. В качестве примеров выбраны Java, Clojure, Elixir и C (в главе о параллельном программировании с применением каркаса OpenCL (*https://en.wikipedia.org/wiki/OpenCL*)). Модель CSP рассматривается на примере кода на Clojure, хотя языку Go следовало бы отдать должное за популяризацию этого подхода. На примере языка Elixir иллюстрируется модель акторов. В свободно доступной альтернативной бонусной главе (*https://media.pragprog.com/titles/pb7con/Bonus_Chapter.pdf*) об акторах используются Scala и каркас Akka. Для тех, кто незнаком со Scala, Elixir является более простым языком для изучения и экспериментов с моделью акторов и платформой распределенных систем Erlang/OTP.

---

[1] *Батчер П.* Семь моделей конкуренции и параллелизма за семь недель. М.: ДМК Пресс, 2015 // *https://dmkpress.com/catalog/computer/programming/functional/978-5-97060-720-6/*

[2] На API пакетов Python `threading` и `concurrent.futures` большое влияние оказала стандартная библиотека Java.

Унмеш Йоши из компании Thoughtworks написал несколько страниц в блог Мартина Фаулера, документирующих «Паттерны распределенных систем». Первая страница (*https://martinfowler.com/articles/patterns-of-distributed-systems/*) содержит великолепное введение в предмет, со ссылками на отдельные паттерны. Йоши добавляет паттерны постепенно, но то, что уже опубликовано, – квинтэссенция добытого тяжким трудом опыта реализации особо ответственных систем.

Книга Martin Kleppmann «Designing Data-Intensive Applications» (O'Reilly) (*https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/*) – редкий образчик книги, написанной практиком с богатым опытом работы в индустрии и хорошей академической подготовкой. Автор работал в компании LinkedIn, имеющей крупномасштабную инфраструктуру данных, и в двух стартапах, после чего занялся научной работой в области распределенных систем в Кембриджском университете. Каждая глава книги Клеппманна заканчивается обширным списком литературы, включая последние научные работы. В книге также имеются многочисленные диаграммы, проясняющие суть дела, и прекрасно исполненные карты концепций.

Мне повезло присутствовать на выдающемся семинаре Франческо Чезарини по архитектуре надежных распределенных систем на конференции OSCON 2016: «Designing and architecting for scalability with Erlang/OTP» (видео (*https://www.oreilly.com/library/view/oscon-2016-video/9781491965153/video247021.html*) имеется на образовательной платформе O'Reilly). Несмотря на название, Чезарини объясняет (отметка 9:35):

> Очень немногое из того, что я хочу сказать, специфично для Erlang [...]. Факт, что Erlang устраняет целый ряд акцидентальных трудностей на пути создания надежных систем, которые никогда не отказывают и при этом масштабируемы. Поэтому ваша задача существенно упростится, если вы будете использовать Erlang или язык, работающий под управлением виртуальной машины Erlang.

Этот семинар основан на последних четырех главах книги Francesco Cesarini, Steve Vinoski «Designing for Scalability with Erlang/OTP[1]» (O'Reilly) (*https://www.oreilly.com/library/view/designing-for-scalability/9781449361556/*).

Программирование распределенных систем – трудное и увлекательное занятие, но помните о *зависти к масштабу веба* (*https://www.thoughtworks.com/radar/techniques/high-performance-envy-web-scale-envy*). Принцип KISS (*https://en.wikipedia.org/wiki/KISS_principle*) – совет, к которому инженерам стоит прислушаться.

Ознакомьтесь со статьей Frank McSherry, Michael Isard, Derek G. Murray «Scalability! But at what COST?» (*https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry*). Авторы идентифицировали системы параллельной обработки графов, представленных на научных симпозиумах, которые требуют сотен ядер, чтобы превзойти по производительности «компетентную однопоточную реализацию». Они также обнаружили системы, которые «работают хуже однопоточных во всех известных конфигурациях».

Эти открытия напоминают мне классическую хакерскую колкость:

> Мой скрипт на Perl работает быстрее твоего Hadoop-кластера.

[1] *Чезарини Ф., Виноски С.* Проектирование масштабируемых систем в Erlang/OTP. М.: ДМК Пресс, 2017 // *https://dmkpress.com/catalog/computer/programming/functional/978-5-97060-212-6/*

Поговорим

Чтобы справляться со сложностью, нужны ограничения

Я учился программировать на калькуляторе TI-58. Его «язык» напоминал ассемблер. На этом уровне все «переменные» были глобальными и такой роскоши, как структурное управление потоком, не было и в помине. Были команды условного перехода, передававшие управление в произвольную точку – до или после текущей команды – в зависимости от значения регистра или флага процессора.

На ассемблере, в общем-то, можно сделать все, что угодно, и в этом проблема: слишком мало ограничений, чтобы предотвратить ошибки и помочь сопровождающим понять код, когда возникает необходимость внести изменения.

Вторым моим языком стал неструктурированный BASIC, стоявший на 8-разрядных компьютерах, – ничего похожего на Visual Basic, появившийся гораздо позже. В нем были предложения `FOR`, `GOSUB` и `RETURN`, но по-прежнему отсутствовало понятие локальных переменных. Предложение `GOSUB` не поддерживало передачу параметров, это было просто пафосное `GOTO`, оно помещало номер строки возврата в стек, так чтобы предложение `RETURN` знало, куда перейти. Подпрограммы должны были получать параметры из глобальных данных и в них же сохранять результаты. Приходилось импровизировать и придумывать другие формы управления потоком с помощью комбинаций `IF` и `GOTO`, которое, как и раньше, позволяло перейти в любую точку программы.

Я помню, с каким трудом, после нескольких лет программирования с помощью переходов и глобальных переменных, я перестраивал мышление на «структурное программирование», когда стал изучать Pascal. Теперь я должен был использовать предложения управления потоком, обрамляющие блоки кода с единственной точкой входа. Я не мог просто перейти в любое нужное мне место. Глобальные переменные – неизбежный атрибут BASIC – теперь оказались под запретом. Я должен был переосмыслить поток данных и явно передавать аргументы функциям.

Следующим вызовом для меня стало изучение объектно-ориентированного программирования. По сути своей объектно-ориентированное программирование – это то же структурное программированиие, но с большим количеством ограничений и полиморфизмом. Сокрытие информации заставляет снова пересмотреть представления о том, где находятся данные. Я помню, сколько раз испытывал горькое разочарование, оказываясь перед необходимостью переделать код, так чтобы мой метод мог получить инкапсулированную в объекте информацию, до которой не мог добраться.

Языки функционального программирования добавляют другие ограничения, однако особенно трудно смириться с неизменяемостью – после десятилетий императивного и объектно-ориентированного программирования. Но, привыкнув к этим ограничениям, мы начинаем рассматривать их как благословение. Они сильно упрощают рассуждения о программе.

Недостаток ограничений – главная проблема модели конкурентного программирования с потоками и блокировками. Подводя итог главе 1 книги «Seven Concurrency Models in Seven Weeks», Пол Бутчер писал:

Однако величайшая слабость этого подхода состоит в том, что программировать с помощью потоков и блокировок *трудно*. Возможно, проектировщику языка и легко добавить их в язык, но нам, бедным программистам, они помогают очень мало.

Приведу несколько примеров ничем не ограниченного поведения в этой модели:

- все потоки могут иметь общий доступ к произвольным изменяемым структурам данных;
- планировщик может прервать поток почти в любой точке, в т. ч. посреди простой операции типа `a += 1`. Очень немногие операции атомарны на уровне выражений в исходном коде;
- блокировки обычно *рекомендательные*. Этот технический термин означает, что мы должны не забыть явно поставить блокировку, перед тем как изменить структуру данных. Если этого не сделать, то ничто не помешает нашему коду создать хаос, пока другой поток честно удерживает блокировку и изменяет те же самые данные.

С другой стороны, рассмотрим некоторые ограничения, налагаемые моделью акторов, в которой единица выполнения называется *актором*[1]:

- у актора может быть внутреннее состояние, но он не может разделять его с другими акторами;
- акторы могут взаимодействовать только путем отправки и получения сообщений;
- сообщения могу содержать лишь копии данных, но не ссылки на изменяемые данные;
- актор обрабатывает только одно сообщение в каждый момент времени. Нет такого понятия, как конкурентное выполнение внутри одного актора.

Конечно, *стиль* кодирования с помощью акторов можно принять в любом языке – нужно только следовать этим правилам. Точно так же можно использовать идиомы объектно-ориентированного программирования в C и даже паттерны структурного программирования в ассемблере. Но это требует множества соглашений и дисциплины со стороны всех причастных к коду.

Управлять блокировками в модели акторов, реализованной в Erlang и Elixir, где все данные неизменяемые, необязательно.

Потоки и блокировки никуда не деваются. Я просто думаю, что неразумно тратить время на работу с такими низкоуровневыми понятиями при написании приложений. К написанию модулей ядра или баз данных это не относится.

[1] В сообществе Erlang для акторов используется термин «процесс». В Erlang каждый процесс является функцией в своем собственном цикле, поэтому они очень мало весят и в каждый момент времени на одной машине могут работать миллионы процессов – ничего общего с тяжеловесными процессами ОС, о которых мы говорили в этой главе. Так что мы здесь имеем примеры обоих грехов, описанных профессором Саймоном: использование разных слов для обозначения одной вещи и использование одного слова для обозначения разных вещей.

Я всегда оставляю за собой право изменить точку зрения. Но в данный момент я убежден, что модель акторов – самая разумная из имеющихся моделей конкурентного программирования общего назначения. Модель CSP (взаимодействующих последовательных процессов) тоже хороша, но ее реализация в Go позволяет обойти некоторые ограничения. Идея CSP заключается в том, что сопрограммы (или *горутины* в Go) обмениваются данными и синхронизируются с помощью очередей (в Go они называются *каналами*). Однако Go поддерживает также разделение памяти и блокировки. Я даже встречал книгу о Go, в которой пропагандируется использование разделяемой памяти и блокировок вместо каналов – во имя производительности. Старые привычки умирают с трудом.

Глава 20

Конкурентные исполнители

> Потоки критикуют в основном системные программисты, имея в виду такие ситуации, с которыми типичный прикладной программист никогда не сталкивается. [...] В 99 % случаев, с которыми имеет дело прикладной программист, достаточно знать, как запустить группу независимых потоков и собрать результаты в очередь.
>
> – Мишель Симионато, вдумчивый пользователь Python[1]

Эта глава посвящена классам `concurrent.futures.Executor`, которые инкапсулируют паттерн «запуска группы независимых потоков и сбора результатов в очередь», описанный Мишелем Симионато. Конкурентные исполнители делают его использование почти тривиальным делом не только с помощью потоков, но и с помощью процессов – что полезно для счетных задач.

Здесь же я введу понятие «*будущего объекта*» – объекта, представляющего асинхронное выполнение операции, по аналогии с обещаниями в JavaScript. Эта плодотворная идея лежит в основе не только библиотеки `concurrent.futures`, но и пакета `asyncio`, рассматриваемого в главе 21.

Что нового в этой главе

Раньше глава называлась «Конкурентность и будущие объекты», теперь я назвал ее «Конкурентные исполнители», потому что именно исполнители – самое важное рассматриваемое в ней высокоуровневое средство. Будущие объекты являются низкоуровневыми объектами, им посвящен раздел «Где находятся будущие объекты?», но в остальной части главы они практически не встречаются.

Во всех примерах HTTP-клиентов теперь используется новая библиотека *HTTPX* (*https://www.python-httpx.org/*), предлагающая как синхронный, так и асинхронный API.

Подготовка сцены для экспериментов в разделе «Загрузка с индикацией хода выполнения и обработкой ошибок» стала проще благодаря многопоточному серверу, добавленному в пакет `http.server` в версии Python 3.7. Раньше в стандартной библиотеке имелся только однопоточный `BaseHttpServer`, которого

[1] Из статьи Мишеля Симионато «Threads, processes and concurrency in Python: some thoughts» (*https://www.artima.com/weblogs/viewpost.jsp?thread=299551*), имеющей подзаголовок «Removing the hype around the multicore (non) revolution and some (hopefully) sensible comment about threads and other forms of concurrency» (Развенчание рекламной чепухи по поводу многоядерной (не) революции и некоторые (надеюсь) полезные замечания о потоках и других видах конкурентности).

было недостаточно для экспериментов с конкурентными клиентами, поэтому в первом издании мне пришлось прибегнуть к внешним инструментам.

В разделе «Запуск процессов с помощью concurrent.futures» теперь показано, как исполнитель упрощает код, который мы видели в разделе «Код проверки на простоту для многоядерной машины» главы 19.

Наконец, большую часть теоретического материала я перенес в главу 19 «Модели конкурентности в Python».

Конкурентная загрузка из веба

Эффективный сетевой ввод-вывод невозможен без конкурентности: чем впустую растрачивать процессорное время на ожидание, лучше заняться чем-то полезным, пока из сети не пришел ответ[1].

Для иллюстрации этого положения я написал три простые программы загрузки изображений флагов 20 стран из веба. Первая программа, *flags.py*, работает последовательно: она запрашивает следующее изображение только после того, как предыдущее загружено и записано на диск. Два других скрипта производят загрузку параллельно: они запрашивают все изображения практически одновременно, а сохраняют по мере поступления. Скрипт *flags_threadpool.py* пользуется пакетом `concurrent.futures`, а *flags_asyncio.py* – пакетом `asyncio`.

В примере 20.1 показаны результаты выполнения всех трех скриптов, по три раза каждый. Я также разместил на YouTube видео продолжительностью 73 с (*https://www.youtube.com/watch?v=A9e9Cy1UkME*), чтобы было видно, как по мере сохранения флагов в окне macOS Finder становятся видны их изображения. Скрипты загружают изображения с сайта *fluentpython.com*, который развернут за системой доставки контента (CDN), так что при первых прогонах они могут работать несколько медленнее. Показанные ниже результаты получены после прогрева кеша CDN.

Пример 20.1. Результаты трех типичных прогонов скриптов flags.py, flags_threadpool.py и flags_asyncio.py

```
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN ❶
20 flags downloaded in 7.26s ❷
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.20s
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.09s
$ python3 flags_threadpool.py
DE BD CN JP ID EG NG BR RU CD IR MX US PH FR PK VN IN ET TR
20 flags downloaded in 1.37s ❸
$ python3 flags_threadpool.py
EG BR FR IN BD JP DE RU PK PH CD MX ID US NG TR CN VN ET IR
20 flags downloaded in 1.60s
$ python3 flags_threadpool.py
```

[1] Особенно если облачный поставщик начисляет плату посекундно вне зависимости от занятости процессоров.

```
BD DE EG CN ID RU IN VN ET MX FR CD NG US JP TR PK BR IR PH
20 flags downloaded in 1.22s
$ python3 flags_asyncio.py
BD BR IN ID TR DE CN US IR PK PH FR RU NG VN ET MX EG JP CD
20 flags downloaded in 1.36s
$ python3 flags_asyncio.py ❹
RU CN BR IN FR BD TR EG VN IR PH CD ET ID NG DE JP PK MX US
20 flags downloaded in 1.27s
$ python3 flags_asyncio.py
RU IN ID DE BR VN PK MX US IR ET EG NG BD FR CN JP PH CD TR ❺
20 flags downloaded in 1.42s
```

❶ Печать результатов каждого прогона начинается с вывода кодов стран в порядке загрузки их флагов и заканчивается сообщением о том, сколько прошло времени.
❷ Скрипту *flags.py* требуется в среднем 7,18 с для загрузки 20 изображений.
❸ Скрипту *flags_threadpool.py* в среднем требуется 1,40 с.
❹ Скрипту *flags_asyncio.py* в среднем требуется 1,35 с.
❺ Обратите внимание на порядок стран: в случае параллельных скриптов загрузка каждый раз происходит в другом порядке.

Между двумя параллельными скриптами разница в производительности несущественна, но тот и другой работают в пять раз быстрее последовательного скрипта – и это на совсем небольшой задаче загрузки 20 файлов размером несколько килобайтов каждый. Если бы количество загружаемых файлов исчислялось сотнями, то параллельные скрипты показали бы рост производительности в 20 и более раз.

При тестировании параллельных HTTP-клиентов в открытом вебе можно случайно организовать DoS-атаку или навлечь на себя такие подозрения. В случае примера 20.1 ничего страшного не случится, потому что в скрипты зашито ограничение: только 20 запросов. Далее в этой главе мы будем использовать для прогона тестов стандартный пакет `http.server`.

Теперь рассмотрим реализации двух скриптов, протестированных в примере 20.1: *flags.py* и *flags_threadpool.py*. Скрипт *flags_asyncio.py* я отложу до главы 21, но продемонстрировать хотел сразу три, чтобы подчеркнуть два момента:

1. Независимо от используемого способа организации конкурентности – многопоточность или сопрограммы – производительность приложения, занятого сетевым вводом-выводом (при правильной реализации), оказывается намного выше, чем у последовательного кода.
2. Для HTTP-клиентов, умеющих контролировать количество отправляемых запросов, разница между потоками и сопрограммами несущественна[1].

Итак, перейдем к коду.

[1] Для серверов, к которым может обращаться много клиентов, разница есть: сопрограммы лучше масштабируются, потому что потребляют гораздо меньше памяти, чем потоки, а также уменьшают стоимость контекстного переключения, о чем я говорил в разделе «Не решение на основе потоков» главы 19.

Скрипт последовательной загрузки

В примере 20.2 показана реализация *flags.py*, первого скрипта, выполненного в примере 20.1. Он не очень интересен, но большая часть его кода и параметров будет использована для реализации конкурентных скриптов, поэтому уделим ему немного внимания.

Для большей ясности в примере 20.2 нет никакой проверки ошибок. Исключениями мы займемся позже, а пока хотим сосредоточиться на структуре кода, чтобы было проще сравнить этот скрипт с конкурентными.

Пример 20.2. flags.py: последовательный скрипт загрузки; некоторые функции будут использованы и в других скриптах

```
import time
from pathlib import Path
from typing import Callable

import httpx ❶

POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split() ❷

BASE_URL = 'https://www.fluentpython.com/data/flags' ❸
DEST_DIR = Path('downloaded') ❹

def save_flag(img: bytes, filename: str) -> None: ❺
    (DEST_DIR / filename).write_bytes(img)

def get_flag(cc: str) -> bytes: ❻
    url = f'{BASE_URL}/{cc}/{cc}.gif'.lower()
    resp = httpx.get(url, timeout=6.1, ❼
                     follow_redirects=True) ❽
    resp.raise_for_status() ❾
    return resp.content

def download_many(cc_list: list[str]) -> int: ❿
    for cc in sorted(cc_list): ⓫
        image = get_flag(cc)
        save_flag(image, f'{cc}.gif')
        print(cc, end=' ', flush=True) ⓬
    return len(cc_list)

def main(downloader: Callable[[list[str]], int]) -> None: ⓭
    DEST_DIR.mkdir(exist_ok=True) ⓮
    t0 = time.perf_counter() ⓯
    count = downloader(POP20_CC)
    elapsed = time.perf_counter() - t0
    print(f'\n{count} downloads in {elapsed:.2f}s')

if __name__ == '__main__':
    main(download_many) ⓰
```

❶ Импортировать библиотеку `httpx`; она не входит в состав стандартной библиотеки, поэтому, по принятому соглашению, импортируется после стандартных модулей, а предложение импорта отделяется пустой строкой.
❷ Список кодов стран (по стандарту ISO 3166) с наибольшим населением, отсортированный в порядке убывания численности населения.
❸ Сайт, откуда загружаются изображения флагов[1].
❹ Локальный каталог, в котором сохраняются изображения.
❺ Скопировать `img` (последовательность байтов) в файл с именем `filename` в каталоге `DEST_DIR`.
❻ Зная код страны, построить URL-адрес и загрузить изображение; вернуть двоичное содержимое ответа.
❼ Считается правильным добавлять разумный тайм-аут для сетевых операций, чтобы избежать ненужной блокировки на несколько минут.
❽ По умолчанию *HTTPX* не выполняет перенаправление[2].
❾ В этом скрипте нет обработки ошибок, но данный метод возбуждает исключение, если состояние HTTP не принадлежит диапазону 2XX. Это рекомендуемая практика, позволяющая избежать «немых» отказов.
❿ `download_many` – основная функция, позволяющая провести сравнение с конкурентными реализациями.
⓫ Обойти список стран в алфавитном порядке, чтобы порядок отображения на выходе был такой же, как на входе; вернуть количество загруженных изображений.
⓬ Отображать по одному коду страны за раз. Все коды отображаются в одной строке, чтобы было видно, как происходит загрузка. Аргумент `end=' '` означает, что обычный символ перевода строки в конце каждой строки нужно заменить пробелом, чтобы все коды стран отображались в одной строке один за другим. Аргумент `flush=True` необходим, потому что по умолчанию Python буферизует выходные строки, т. е. напечатанные символы отображаются только после вывода символа перевода строки.
⓭ При вызове `main` необходимо указывать функцию, которая производит загрузку; таким образом, `main` можно будет использовать как библиотечную функцию, способную работать и с другими реализациями `download_many` в примерах `threadpool` и `ascyncio`.
⓮ Создать каталог `DEST_DIR`, если необходимо; не возбуждать исключение, если каталог уже существует.
⓯ Запомнить и вывести истекшее время после завершения функции загрузки.
⓰ Вызвать `main`, передав ей функцию `download_many`.

---

[1] Оригиналы изображений взяты из мировой книги фактов ЦРУ (*http://1.usa.gov/1JIsmHJ*), общедоступного сайта правительства США. Я скопировал их на свой сайт во избежание непреднамеренной DoS-атаки на сайт *cia.gov*.

[2] В этом примере задавать `follow_redirects=True` необязательно, но я хотел подчеркнуть важное различие между библиотеками *HTTPX* и *requests*. Кроме того, задание `follow_redirects=True` позволяет мне разместить файлы изображений в другом месте, если в будущем возникнет такая необходимость. Я думаю, что подразумеваемое по умолчанию в *HTTPX* значение `follow_redirects=False` разумно, потому что неожиданное перенаправление может замаскировать лишние запросы и усложнить диагностику ошибок.

Источником вдохновения при написании библиотеки *HTTPX* (*https://www.python-httpx.org/*) стал пакет *requests*, но она построена на более современном фундаменте. Особенно важно, что *HTTPX* предлагает синхронный и асинхронный API, поэтому мы можем использовать ее во всех примерах HTTP-клиентов в этой и следующей главах. Стандартная библиотека Python содержит модуль `urllib.request`, но у него есть только синхронный API, да и тот не особенно дружелюбен к пользователю.

Ничего особенно нового в скрипте *flags.py* нет. Он служит просто эталоном для сравнения с другими скриптами, а использую я его как библиотеку, чтобы не писать лишний код. Теперь рассмотрим другую реализацию – на основе библиотеки `concurrent.futures`.

Загрузка с применением библиотеки concurrent.futures

Основой пакета `concurrent.futures` являются классы `ThreadPoolExecutor` и `ProcessPoolExecutor`, которые реализуют API, позволяющий передавать вызываемые объекты соответственно потокам или процессам. Оба класса прозрачно управляют внутренним пулом рабочих потоков или процессов и очередью подлежащих выполнению задач. Но поскольку интерфейс высокоуровневый, нам не нужно знать об этих деталях для такого простого дела, как загрузка флагов.

В примере 20.3 показан простейший способ параллельной загрузки – методом `ThreadPoolExecutor.map`.

Пример 20.3. flags_threadpool.py: многопоточный скрипт загрузки с применением класса `futures.ThreadPoolExecutor`

```
from concurrent import futures

from flags import save_flag, get_flag, main ❶

def download_one(cc: str): ❷
    image = get_flag(cc)
    save_flag(image, f'{cc}.gif')
    print(cc, end=' ', flush=True)
    return cc

def download_many(cc_list: list[str]) -> int:
    with futures.ThreadPoolExecutor() as executor: ❸
        res = executor.map(download_one, sorted(cc_list)) ❹

    return len(list(res)) ❺

if __name__ == '__main__':
    main(download_many) ❻
```

❶ Использовать некоторые функции из модуля `flags` (пример 20.2).

❷ Функция, загружающая одно изображение; ее будет исполнять каждый поток.

❸ Создать экземпляр `ThreadPoolExecutor` как контекстный менеджер; метод `executor.__exit__` вызовет `executor.shutdown(wait=True)`, который блокирует выполнение программы до завершения всех потоков.

❹ Метод `map` похож на встроенную функцию `map` с тем исключением, что функция `download_one` конкурентно вызывается из нескольких потоков; он возвращает генератор, который можно обойти для получения значений, возвращенных каждой функцией, – в данном случае каждое обращение к `download_one` возвращает код страны.

❺ Вернуть количество полученных результатов. Если функция в каком-то потоке возбудила исключение, то оно возникнет в этом месте, когда неявный вызов `next()` из конструктора `list` попытается получить соответствующее значение от итератора, возвращенного методом `.map`.

❻ Вызвать функцию `main` из модуля `flags`, передавая ей конкурентную версию `download_many`.

Отметим, что функция `download_one` из примера 20.3, по сути дела, является телом цикла `for` в функции `download_many` из примера 20.2. Это типичный рефакторинг, встречающийся при написании конкурентного кода: преобразовать тело последовательного цикла `for` в функцию, которая будет вызываться конкурентно.

Пример 20.3 получился очень коротким, потому что мне удалось повторно использовать большинство функций из последовательного скрипта *flags.py*. Одна из самых замечательных особенностей `concurrent.futures` – то, как просто можно добавить конкурентное выполнение поверх унаследованного последовательного кода.

Конструктор `ThreadPoolExecutor` принимает несколько аргументов, но первым и самым важным является `max_workers`, который задает максимальное число исполняемых потоков. Если `max_workers` равно `None` (по умолчанию), то `ThreadPoolExecutor` вычисляет значение по формуле (начиная с версии Python 3.8):

```
max_workers = min(32, os.cpu_count() + 4)
```

Обоснование приводится в документации по `ThreadPoolExecutor` (*https://docs.python.org/3.10/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor*):

> Это значение по умолчанию оставляет как минимум 5 исполнителей для задач ввода-вывода. Оно позволяет задействовать не более 32 процессорных ядер для счетных задач, освобождающих GIL. А это позволяет избежать чрезмерного потребления ресурсов на многократноядерных машинах.
>
> Теперь класс `ThreadPoolExecutor` повторно использует простаивающие рабочие потоки, прежде чем запускать `max_workers` исполнителей.

Итак, вычисленное значение `max_workers` разумно, и `ThreadPoolExecutor` не запускает новые рабочие потоки без необходимости. Понимание логики, стоящей за вычислением `max_workers`, поможет вам решить, когда и как устанавливать это значение самостоятельно.

Библиотека называется `concurrency.futures`, но в примере 20.3 мы никаких «futures» не видели. Возникает законный вопрос: где же они? Ответ дан в следующем разделе.

Где находятся будущие объекты?

Будущие объекты – важнейшие компоненты внутреннего механизма пакетов `concurrent.futures` и `asyncio`, но не всегда они видны пользователям этих библиотек. В примере 20.3 будущие объекты используются за кулисами, но мой код напрямую к ним не обращается. В этом разделе сообщаются общие сведения о будущих объектах, с примером их практического применения.

Начиная с версии Python 3.4 в стандартной библиотеке есть два класса с именем `Future`: `concurrent.futures.Future` и `asyncio.Future`. Они служат одной и той же цели: экземпляр класса `Future` представляет некое отложенное вычисление, завершившееся или нет. Это аналог класса `Deferred` в Twisted, класса `Future` в Tornado и объектов `Promise` в современном JavaScript.

Будущие объекты инкапсулируют ожидающие операции, так что их можно помещать в очереди, опрашивать состояние завершения и получать результаты (или исключения), когда они станут доступны.

Важно понимать, что ни вы, ни я не должны создавать будущие объекты: предполагается, что их создает исключительно используемая библиотека, будь то `concurrent.futures` или `asyncio`. Легко понять, почему это так: объект `Future` представляет нечто, что должно случиться когда-то в будущем, а единственный способ гарантировать, что это действительно случится, – запланировать выполнение объекта. Поэтому экземпляры класса `concurrent.futures.Future` создаются только в результате планирования выполнения какой-то операции с помощью одного из подклассов `concurrent.futures.Executor`. Например, метод `Executor.submit()` принимает вызываемый объект, планирует его выполнение и возвращает будущий объект.

Прикладной код не должен изменять состояние будущего объекта: его изменит каркас конкурентности, когда представляемое этим объектом вычисление завершится, а мы не можем управлять тем, когда это произойдет.

Оба класса `Future` имеют неблокирующий метод `.done()`, который возвращает булево значение, показывающее, завершился вызываемый объект, связанный с экземпляром этого класса, или нет. Но вместо того чтобы раз за разом интересоваться, не завершил ли работу будущий объект, клиент обычно просит, чтобы его уведомили. Поэтому в обоих классах `Future` имеется метод `.add_done_callback()`: если передать ему вызываемый объект, то он будет вызван, когда будущий объект завершится, а в качестве единственного аргумента будет передан сам этот будущий объект. Имейте в виду, что вызываемый объект работает в том же потоке или процессе, что и функция, обернутая будущим объектом.

Существует также метод `.result()`, который одинаково работает в обоих классах в ситуации, когда выполнение будущего объекта завершено: либо возвращает результат вызываемого объекта, либо повторно возбуждает исключение, возникшее во время выполнения. Но если выполнение будущего объекта еще не завершено, то метод `result` ведет себя совершенно по-разному. В объекте класса `concurrency.futures.Future` вызов `f.result()` блокирует вызывающий поток до тех пор, пока не будет готов результат. Если передан необязательный аргумент `timeout` и выполнение будущего объекта не завершилось в отведенное время, то возбуждается исключение `TimeoutError`. Метод `asyncio.Future.result` не под-

держивает задание тайм-аута, а рекомендуемый способ получения результата будущего объекта заключается в использовании `await` – к объектам класса `concurrency.futures.Future` этот подход неприменим.

Будущие объекты возвращаются несколькими функциями из обеих библиотек; другие пользуются ими внутри себя, невидимо для пользователя. Примером второго рода может служить функция `Executor.map`, с которой мы встречались в примере 20.3: она возвращает итератор, метод `__next__` которого вызывает метод `result` каждого будущего объекта, так что мы получаем не сами будущие объекты, а результаты их выполнения.

Чтобы попрактиковаться в использовании будущих объектов, перепишем пример 17.3 с использованием функции `concurrent.futures.as_completed`, которая принимает итерируемый объект, содержащий будущие объекты, и возвращает итератор, который отдает будущие объекты по мере их выполнения.

Чтобы можно было воспользоваться функцией `futures.as_completed`, необходимо внести изменения только в функцию `download_many`. Вызов высокоуровневого метода `executor.map` заменяется двумя циклами `for`: один – для создания и планирования будущих объектов, другой – для получения их результатов. И заодно уж добавим несколько вызовов `print` для печати каждого будущего объекта до и после завершения. В примере 20.4 показан код новой функции `download_many`. Количество строк в ней увеличилось с 5 до 17, зато теперь можно присмотреться к таинственным будущим объектам. Все остальные функции такие же, как в примере 20.3.

Пример 20.4. flags_threadpool_futures.py: замена `executor.map` на `executor.submit` и `futures.as_completed` в функции `download_many`

```
def download_many(cc_list: list[str]) -> int:
    cc_list = cc_list[:5] ❶
    with futures.ThreadPoolExecutor(max_workers=3) as executor: ❷
        to_do: list[futures.Future] = []
        for cc in sorted(cc_list): ❸
            future = executor.submit(download_one, cc) ❹
            to_do.append(future) ❺
            print(f'Scheduled for {cc}: {future}') ❻

        for count, future in enumerate(futures.as_completed(to_do), 1): ❼
            res: str = future.result() ❽
            print(f'{future} result: {res!r}') ❾
    return count
```

❶ Для этой демонстрации мы ограничимся только пятью странами с самой большой численностью населения.

❷ Установить значение `max_workers` равным 3, чтобы можно было следить за ожидающими будущими объектами в распечатке.

❸ Обойти коды стран в алфавитном порядке, чтобы было понятно, что результаты поступают не по порядку.

❹ Метод `executor.submit` планирует выполнение вызываемого объекта и возвращает объект `future`, представляющий ожидаемую операцию.

❺ Сохранить каждый будущий объект, чтобы впоследствии его можно было извлечь с помощью функции `as_completed`.

❻ Вывести сообщение, содержащее код страны и соответствующий ему будущий объект `future`.
❼ `as_completed` отдает будущие объекты по мере их завершения.
❽ Получить результат этого объекта `future`.
❾ Отобразить объект `future` и результат его выполнения.

Отметим, что вызов `future.result()` в этом примере никогда не приводит к блокировке, потому что будущий объект получен как результат `as_completed`. В примере 20.5 показан результат одного прогона программы из примера 20.4.

Пример 20.5. Результат работы скрипта flags_threadpool_futures.py

```
$ python3 flags_threadpool_ac.py
Scheduled for BR: <Future at 0x100791518 state=running> ❶
Scheduled for CN: <Future at 0x100791710 state=running>
Scheduled for ID: <Future at 0x100791a90 state=running>
Scheduled for IN: <Future at 0x101807080 state=pending>  ❷
Scheduled for US: <Future at 0x101807128 state=pending>
CN <Future at 0x100791710 state=finished returned str> result: 'CN'  ❸
BR ID <Future at 0x100791518 state=finished returned str> result: 'BR'  ❹
<Future at 0x100791a90 state=finished returned str> result: 'ID'
IN <Future at 0x101807080 state=finished returned str> result: 'IN'
US <Future at 0x101807128 state=finished returned str> result: 'US'

5 downloads in 0.70s
```

❶ Будущие объекты планируются в алфавитном порядке; метод `repr()` будущего объекта показывает его состояние: первые три объекта выполняются, поскольку есть всего три рабочих потока.
❷ Последние два будущих объекта ожидают освобождения рабочего потока.
❸ Первое слово `CN` напечатано функцией `download_one`, исполняемой в рабочем потоке, остаток строки напечатан функцией `download_many`.
❹ Здесь два потока выводят коды стран, прежде чем `download_many` в главном потоке получает возможность вывести результат объекта в первом потоке.

Я рекомендую поэкспериментировать со скриптом *flags_threadpool_futures.py*. Если прогнать его несколько раз подряд, то мы увидим, что порядок вывода результатов изменяется. При увеличении `max_workers` до 5 изменчивость порядка усиливается, а при уменьшении до 1 код начинает работать последовательно, и результаты выводятся в том же порядке, в каком коды стран подавались методом `submit`.

Мы видели два варианта скрипта загрузки с применением библиотеки `concurrent.futures`: пример 20.3 на основе метода `ThreadPoolExecutor.map` и пример 20.4 на основе `futures.as_completed`. Если вам не терпится увидеть код скрипта *flags_asyncio.py*, можете взглянуть на пример 21.3 в главе 21.

А теперь посмотрим, как с помощью `concurrent.futures` можно просто обойти GIL для счетных задач.

Запуск процессов с помощью concurrent.futures

Страница документации по пакету `concurrent.futures` (*https://docs.python.org/3/library/concurrent.futures.html*) имеет подзаголовок «Запуск параллельных задач». Этот пакет поддерживает параллельные вычисления на многоядерных машинах, потому что умеет распределять работу между несколькими процессами Python благодаря классу `ProcessPoolExecutor`.

И `ProcessPoolExecutor`, и `ThreadPoolExecutor` реализуют обобщенный интерфейс `Executor` (*https://docs.python.org/3.10/library/concurrent.futures.html#concurrent.futures.Executor*), поэтому, работая с `concurrent.futures`, очень легко переходить от решения на основе потоков к решению на основе процессов и обратно.

Использование `ProcessPoolExecutor` не дает никакого преимущества в примере загрузки флагов или в любой другой программе, ограниченной скоростью ввода-вывода. И это легко проверить – просто измените следующие строки в примере 20.3:

```
    def download_many(cc_list: list[str]) -> int:
        with futures.ThreadPoolExecutor() as executor:
```

на такие:

```
    def download_many(cc_list: list[str]) -> int:
        with futures.ProcessPoolExecutor() as executor:
```

Конструктор `ProcessPoolExecutor` принимает также аргумент `max_workers`, по умолчанию равный `None`. В данном случае исполнитель ограничивает число рабочих процессов величиной, возвращаемой функцией `os.cpu_count()`.

Процессы потребляют больше памяти и запускаются дольше, чем потоки, поэтому ценность `ProcessPoolExecutor` становится очевидной только для счетных задач. Давайте вернемся к примеру проверки чисел на простоту из раздела «Доморощенный пул процессов» главы 19 и перепишем его с помощью `concurrent.futures`.

И снова о проверке на простоту на многоядерной машине

В разделе «Код проверки на простоту для многоядерной машины» главы 19 мы изучали скрипт *procs.py*, который проверял большие числа на простоту с помощью пакета `multiprocessing`. В примере 20.6 та же задача решается с помощью скрипта *proc_pool.py*, в котором используется объект `ProcessPoolExecutor`. От первого предложения импорта и до вызова `main()` в конце скрипт *procs.py* насчитывает 43 непустые строки кода, а *proc_pool.py* только 31 – на 28 % короче.

Пример 20.6. proc_pool.py: procs.py, переписанный с использованием `ProcessPoolExecutor`

```
import sys
from concurrent import futures ❶
from time import perf_counter
from typing import NamedTuple

from primes import is_prime, NUMBERS

class PrimeResult(NamedTuple): ❷
    n: int
```

```
    flag: bool
    elapsed: float

def check(n: int) -> PrimeResult:
    t0 = perf_counter()
    res = is_prime(n)
    return PrimeResult(n, res, perf_counter() - t0)

def main() -> None:
    if len(sys.argv) < 2:
        workers = None ❸
    else:
        workers = int(sys.argv[1])

    executor = futures.ProcessPoolExecutor(workers) ❹
    actual_workers = executor._max_workers # type: ignore ❺

    print(f'Checking {len(NUMBERS)} numbers with {actual_workers} processes:')

    t0 = perf_counter()
    numbers = sorted(NUMBERS, reverse=True) ❻
    with executor: ❼
        for n, prime, elapsed in executor.map(check, numbers): ❽
            label = 'P' if prime else ' '
            print(f'{n:16} {label} {elapsed:9.6f}s')

    time = perf_counter() - t0
    print(f'Total time: {time:.2f}s')

if __name__ == '__main__':
    main()
```

❶ Нет нужды импортировать `multiprocessing`, `SimpleQueue` и т. д., потому что `concurrent.futures` скрывает все это.

❷ Кортеж `PrimeResult` и функция `check` такие же, как в скрипте *procs.py*, но очереди и функция `worker` больше не нужны.

❸ Вместо того чтобы самостоятельно решать, сколько рабочих процессов использовать, когда их количество в командной строке не задано, мы присваиваем переменной `workers` значение `None` и отдаем решение на усмотрение `ProcessPoolExecutor`.

❹ Здесь я создаю `ProcessPoolExecutor` раньше блока `with` в точке ❼, чтобы в следующей строке можно было напечатать фактическое количество рабочих процессов.

❺ Переменная `_max_workers` – недокументированный атрибут экземпляра в классе `ProcessPoolExecutor`. Я решил использовать его, чтобы показать количество рабочих процессов, когда переменная `workers` равна `None`. Муру ругается – и правильно, – когда я обращаюсь к ней, поэтому я поставил `type: ignore comment`, чтобы утихомирить ее.

❻ Отсортировать подлежащие проверке числа в порядке убывания. Это позволит высветить разницу в поведении *proc_pool.py* и *procs.py*. См. объяснение после данного примера.

❼ Использовать `executor` как контекстный менеджер.

❽ Вызов `executor.map` возвращает экземпляры `PrimeResult`, полученные от функции `check`, в таком же порядке, как аргументы `numbers`.

Выполнив пример 20.6, вы увидите, что результаты появляются строго в порядке убывания, как показано в примере 20.7. Напротив, на порядок результатов в *procs.py* (показан в разделе «Решение на основе процессов» главы 19) сильно влияет трудность проверки конкретного числа на простоту. Например, *procs.py* показывает результат для 7777777777777777 в начале, потому что у этого числа есть малый множитель 7, поэтому `is_prime` быстро определяет, что число не простое.

С другой стороны, 7777777536340681 равно 88191709^2, поэтому `is_prime` требуется гораздо больше времени, чтобы убедиться в том, что это составное число, и еще больше времени, чтобы понять, что число 7777777777777753 простое, – поэтому оба этих числа находятся в конце списка, напечатанного *procs.py*.

Запустив *proc_pool.py*, вы увидите еще одну вещь – кажется, что программа зависла, напечатав результат для 9999999999999999.

Пример 20.7. Вывод proc_pool.py

```
$ ./proc_pool.py
Checking 20 numbers with 12 processes:
9999999999999999   0.000024s ❶
9999999999999917 P 9.500677s ❷
7777777777777777   0.000022s ❸
7777777777777753 P 8.976933s
7777777536340681   8.896149s
6666667141414921   8.537621s
6666666666666719 P 8.548641s
6666666666666666   0.000002s
5555555555555555   0.000017s
5555555555555503 P 8.214086s
5555553133149889   8.067247s
4444444488888889   7.546234s
4444444444444444   0.000002s
4444444444444423 P 7.622370s
3333335652092209   6.724649s
3333333333333333   0.000018s
3333333333333301 P 6.655039s
 299593572317531 P 2.072723s
 142702110479723 P 1.461840s
               2 P 0.000001s
Total time: 9.65s
```

❶ Эта строка появляется очень быстро.
❷ Перед печатью строки проходит более 9,5 с.
❸ Остальные строки появляются почти сразу.

Объясним, почему *proc_pool.py* ведет себя таким образом.

❍ Как уже отмечалось, `executor.map(check, numbers)` возвращает результаты в том же порядке, в каком заданы числа `numbers`.
❍ По умолчанию *proc_pool.py* использует столько рабочих процессов, сколько имеется процессоров, – именно так ведет себя `ProcessPoolExecutor`, когда `max_workers` равно `None`. На моем ноутбуке запущено 12 процессов.

- Поскольку мы подаем `numbers` в порядке убывания, первое число равно 9999999999999999; у него есть делитель 9, так что проверка завершается быстро.
- Второе число равно 9999999999999917, это самое большое простое число в нашей выборке. На его проверку уходит больше времени, чем на проверку любого другого числа.
- Тем временем остальные 11 процессов занимаются проверкой других чисел, которые являются либо простыми, либо составными с большими множителями, либо составными с очень малыми множителями.
- Когда процесс, отвечающий за число 9999999999999917, наконец определит, что оно простое, все остальные процессы уже завершили работу, поэтому результаты появляются немедленно.

Хотя продвижение скрипта *proc_pool.py* не так наглядно, как в случае *procs.py*, общее время выполнения практически такое же, как на рис. 19.2, при том же числе рабочих процессов и процессорных ядер.

Понять поведение конкурентных программ непросто, поэтому я опишу второй эксперимент, который поможет наглядно представить, как работает `Executor.map`.

Эксперименты с Executor.map

Изучим метод `Executor.map`, на этот раз воспользовавшись классом `ThreadPoolExecutor` с тремя рабочими потоками, исполняющими пять вызываемых объектов, которые выводят сообщения с временными метками. Код показан в примере 20.8, а результат в примере 20.9.

Пример 20.8. demo_executor_map.py: простая демонстрация метода `map` объекта `ThreadPoolExecutor`

```
from time import sleep, strftime
from concurrent import futures

def display(*args): ❶
    print(strftime('[%H:%M:%S]'), end=' ')
    print(*args)

def loiter(n): ❷
    msg = '{}loiter({}): doing nothing for {}s...'
    display(msg.format('\t'*n, n, n))
    sleep(n)
    msg = '{}loiter({}): done.'
    display(msg.format('\t'*n, n))
    return n * 10

def main(): ❸
    display('Script starting.')
    executor = futures.ThreadPoolExecutor(max_workers=3) ❹
    results = executor.map(loiter, range(5)) ❺
    display('results:', results) ❻
```

```
    display('Waiting for individual results:')
    for i, result in enumerate(results): ❼
        display(f'result {i}: {result}')

if __name__ == '__main__':
    main()
```

❶ Эта функция печатает переданные ей аргументы, добавляя временную метку в формате `[HH:MM:SS]`.

❷ Функция `loiter` печатает время начала работы, затем спит *n* секунд и печатает время окончания; знаки табуляции формируют отступ сообщения в соответствии с величиной *n*.

❸ `loiter` возвращает `n * 10`, чтобы нагляднее представить результаты.

❹ Создать объект `ThreadPoolExecutor` с тремя потоками.

❺ Передать исполнителю `executor` пять задач (поскольку есть только три потока, сразу начнут выполнение лишь три из них: вызывающие `loiter(0)`, `loiter(1)` и `loiter(2)`); это неблокирующий вызов.

❻ Немедленно распечатать объект `results`, полученный от `executor.map`: это генератор, как видно из результатов, показанных в примере 20.9.

❼ Обращение к `enumerate` в цикле `for` неявно вызывает функцию `next(results)`, которая, в свою очередь, вызывает метод `_f.result()` (внутреннего) будущего объекта `_f`, представляющего первый вызов, `loiter(0)`. Метод `result` блокирует программу до завершения будущего объекта, поэтому каждая итерация этого цикла будет ждать готовности следующего результата.

Призываю вас прогнать пример 20.8 и полюбоваться на то, как постепенно печатаются сообщения. А заодно уж поэкспериментируйте с аргументом `max_workers` объекта `ThreadPoolExecutor` и с функцией `range`, которая порождает аргументы для обращения к `executor.map`, – или замените ее списками подобранных вручную значений, если хотите задать другие задержки.

В примере 20.9 показаны результаты прогона программы из примера 20.8.

Пример 20.9. Результаты прогона скрипта demo_executor_map.py из примера 20.8

```
$ python3 demo_executor_map.py
[15:56:50] Script starting.  ❶
[15:56:50] loiter(0): doing nothing for 0s... ❷
[15:56:50] loiter(0): done.
[15:56:50]      loiter(1): doing nothing for 1s... ❸
[15:56:50]              loiter(2): doing nothing for 2s...
[15:56:50] results: <generator object result_iterator at 0x106517168> ❹
[15:56:50]                      loiter(3): doing nothing for 3s... ❺
[15:56:50] Waiting for individual results:
[15:56:50] result 0: 0 ❻
[15:56:51]     loiter(1): done. ❼
[15:56:51]                              loiter(4): doing nothing for 4s...
[15:56:51] result 1: 10 ❽
[15:56:52]              loiter(2): done. ❾
[15:56:52] result 2: 20
[15:56:53]                      loiter(3): done.
[15:56:53] result 3: 30
[15:56:55]                              loiter(4): done. ❿
[15:56:55] result 4: 40
```

❶ Прогон начался в 15:56:50.
❷ Первый поток выполняет `loiter(0)`, поэтому спит 0 с и завершается еще до того, как второй поток запустился, но на вашей машине все может быть по-другому[1].
❸ `loiter(1)` и `loiter(2)` запускаются немедленно (поскольку в пуле три рабочих потока, он может одновременно выполнять три функции).
❹ Отсюда видно, что объект `results`, возвращенный `executor.map`, – генератор; до сих пор никаких блокировок не было вне зависимости от количества задач и значения `max_workers`.
❺ Поскольку `loiter(0)` завершилась, первый рабочий поток готов к выполнению `loiter(3)`.
❻ Здесь выполнение может быть заблокировано в зависимости от параметров `loiter`: метод `__next__` генератора `results` должен дождаться завершения первого будущего объекта. В данном случае блокировки не будет, потому что вызов `loiter(0)` завершился еще до начала цикла. Отметим, что все действия до этого места произошли в течение одной секунды: 15:56:50.
❼ `loiter(1)` завершается в следующую секунду – в 15:56:51. Поток освобождается и готов к выполнению `loiter(4)`.
❽ Показан результат `loiter(1)`: 10. Теперь цикл `for` блокируется в ожидании результата `loiter(2)`.
❾ Картина повторяется: `loiter(2)` завершается и печатается его результат; затем то же самое для `loiter(3)`.
❿ `loiter(4)` завершается после двухсекундной задержки, поскольку началась в 15:56:51 и ничего не делала 4 с.

Функцией `executor.map` пользоваться легко, но зачастую желательно получать результаты по мере готовности вне зависимости от порядка подачи исходных данных. Для этого нужна комбинация метода `executor.submit` и функции `futures.as_completed`, которую мы видели в примере 20.4. Мы вернемся к этой технике в разделе «Использование `futures.as_completed`» ниже.

Комбинация `executor.submit` и `futures.as_completed` обладает большей гибкостью, чем `executor.map`, потому что ей можно подавать различные вызываемые объекты и аргументы, тогда как `executor.map` предназначена для выполнения одного и того же вызываемого объекта с разными аргументами. Кроме того, множество будущих объектов, передаваемых `futures.as_completed`, может поступать от нескольких исполнителей – одни из них могли быть созданы экземпляром `ThreadPoolExecutor`, другие – экземпляром `ProcessPoolExecutor`.

В следующем разделе мы вернемся к программам загрузки флагов, но предъявим новые требования, которые заставят нас обходить результаты `futures.as_completed` вместо использования `executor.map`.

[1] С потоками никогда не знаешь точную последовательность событий, которые должны произойти практически одновременно; вполне возможно, что на другой машине `loiter(1)` начнется раньше, чем `loiter(0)` завершится, особенно если учесть, что `sleep` всегда освобождает GIL, так что Python может переключиться на другой поток, пусть даже текущий спал 0 с.

Загрузка с индикацией хода выполнения и обработкой ошибок

Как уже отмечалось, в скриптах из раздела «Пример: три способа загрузки из веба» нет обработки ошибок. Это сделано для того, чтобы их было проще читать и сравнивать три подхода: последовательный, многопоточный и асинхронный.

Для тестирования обработки различных ошибок я создал следующие скрипты:

flags2_common.py

Этот модуль содержит общие функции и параметры, используемые во всех flags2-скриптах, в том числе функцию `main`, которая занимается разбором командной строки, хронометражем и выводом результатов. Это чисто вспомогательный модуль, не имеющий прямого отношения к теме главы, поэтому я не стал помещать здесь исходный код, но вы можете найти его на сопроводительном сайте в репозитории *fluentpython/example-code-2e*: файл *20-executors/getflags/flags2_common.py*.

flags2_sequential.py

Последовательный HTTP-клиент с корректной обработкой ошибок и индикацией хода выполнения. Функция `download_one` из этого модуля используется также в скрипте *flags2_threadpool.py*.

flags2_threadpool.py

Конкурентный HTTP-клиент, основанный на классе `futures.ThreadPoolExecutor`; демонстрирует обработку ошибок и интеграцию с индикатором хода выполнения.

flags2_asyncio.py

Та же функциональность, что в предыдущем примере, но на основе `asyncio` и `aiohttp`. Будет рассмотрен в разделе «Улучшение скрипта загрузки на основе asyncio» главы 21.

Будьте осторожны при тестировании параллельных клиентов

При тестировании параллельных HTTP-клиентов, обращающихся к публичным HTTP-серверам, количество запросов в секунду может быть довольно велико, а это признак DoS-атаки. Искусственно ограничивайте производительность клиентов, посылающих запросы публичным серверам. Для тестирования настройте свой локальный HTTP-сервер. Инструкции о том, как это сделать, приведены в разделе «Настройка тестовых серверов» ниже.

Самое заметное визуальное отличие `flags2`-скриптов состоит в том, что они выводят анимированный текстовый индикатор хода выполнения, реализованный с помощью пакета *tqdm* (*https://github.com/noamraph/tqdm*). Я разместил на YouTube ролик продолжительностью 108 с (*https://www.youtube.com/watch?v=M8Z65tAl5l4*), в котором показан индикатор и сравнивается скорость работы всех трех скриптов. В этом ролике я сначала запустил последователь-

ный загрузчик, но через 32 с прервал его, потому что для обращения к 676 URL-адресам и загрузки 194 флагов ему требуется больше 5 минут. Затем я по три раза прогнал многопоточный и асинхронный скрипты, и всякий раз они завершали работу не более чем за 6 с (т. е. в 60 с лишним раз быстрее). На рис. 20.1 показаны два снимка экрана: во время и после работы *flags2_threadpool.py*.

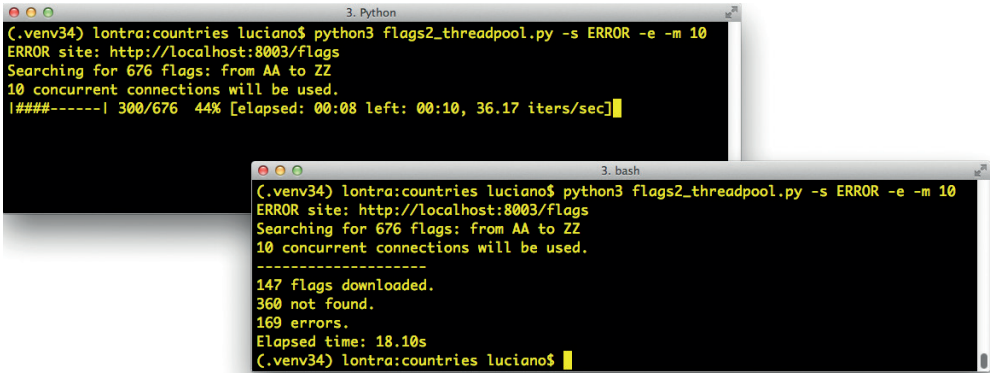


Рис. 20.1. Слева вверху: скрипт flags2_threadpool.py с динамическим индикатором хода выполнения, созданным с помощью tqdm. Справа внизу: то же окно терминала после завершения скрипта

Простейший пример использования *TQDM* приведен в анимированном GIF-файле в файле проекта *README.md* (*https://github.com/noamraph/tqdm/blob/master/README.md*). Установив пакет *tqdm* и набрав следующий код в оболочке Python, вы увидите анимированный индикатор хода выполнения на месте комментария:

```
>>> import time
>>> from tqdm import tqdm
>>> for i in tqdm(range(1000)):
...     time.sleep(.01)
...
>>> # -> здесь будет индикатор хода выполнения <-
```

Помимо зрительно приятного эффекта, функция `tqdm` интересна и с концептуальной точки зрения: она принимает произвольный итерируемый объект и порождает итератор, при обходе которого отображаются индикатор хода выполнения и оценка времени, оставшегося до завершения всех итераций. Чтобы вычислить эту оценку, `tqdm` должна получать либо итерируемый объект, имеющий метод `len`, либо второй аргумент, который содержит ожидаемое количество элементов. Включение `tqdm` в наши `flags2`-примеры дает возможность ближе познакомиться с внутренним устройством параллельных скриптов, поскольку заставляет использовать функции `futures.as_completed` (*https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.as_completed*) и `asyncio.as_completed` (*https://docs.python.org/3/library/asyncio-task.html#asyncio.as_completed*), чтобы `tqdm` могла показать индикатор в момент завершения каждого будущего объекта.

Еще одна особенность `flags2`-примеров – интерфейс командной строки. Все три скрипта принимают одни и те же параметры, а чтобы увидеть их, нужно запустить скрипт с флагом `-h`. Текст справки показан в примере 20.10.

Пример 20.10. Справка для скриптов из серии flags2

```
$ python3 flags2_threadpool.py -h
usage: flags2_threadpool.py [-h] [-a] [-e] [-l N] [-m CONCURRENT] [-s LABEL]
                            [-v]
                            [CC [CC ...]]

Загружает флаги стран с указанными кодами. По умолчанию: 20 стран с
наибольшим населением.

Позиционные аргументы:
CC                  код страны или первая буква (например, B вместо BA...BZ)

Необязательные аргументы:
-h, --help          вывести это сообщение и выйти
-a, --all           вывести все имеющиеся флаги (от AD до ZW)
-e, --every         вывести флаги для всех возможных кодов (AA...ZZ)
-l N, --limit N     ограничиться первыми N кодами
-m CONCURRENT, --max_req CONCURRENT
                    максимальное число параллельных запросов (по умолчанию 30)
-s LABEL, --server LABEL
                    тип сервера: DELAY, ERROR, LOCAL, REMOTE
                    (по умолчанию LOCAL)
-v, --verbose       выводить подробную информацию о ходе выполнения
```

Все аргументы необязательны. Но параметр `-s/--server` игнорировать не следует: он позволяет задать тип и URL-адрес HTTP-сервера, к которому будет обращаться скрипт. Можно задать одну из четырех строк (регистр не важен):

`LOCAL`

Использовать адрес `http://localhost:8000/flags`; это значение по умолчанию. Локальный HTTP-сервер следует настроить так, чтобы он отвечал на запросы к порту 8000. См. инструкции в следующем замечании.

`REMOTE`

Использовать `http://fluentpython.com/data/flags`; это принадлежащий мне публичный сайт, размещенный на разделяемом сервере. Не бомбардируйте его слишком большим количеством параллельных запросов. Домен *fluentpython.com* связан с бесплатной учетной записью в Cloudflare CDN (*http://www.cloudflare.com/*), так что первые загрузки могут оказаться довольно медленными, но скорость возрастет по мере прогрева кеша CDN.

`DELAY`

Использовать `http://localhost:8001/flags`; прокси-сервер, задерживающий HTTP-ответы, должен прослушивать порт 8001. Чтобы было проще экспериментировать, я написал скрипт *slow_server.py*. Вы можете найти его в каталоге *20-futures/getflags/* репозитория кода (*https://github.com/fluentpython/example-code-2e*). См. инструкции в следующем замечании.

`ERROR`

Использовать `http://localhost:8002/flags`; сервер, отправляющий коды ошибок HTTP, должен прослушивать порт 8002. Инструкции см. ниже

Настройка тестовых серверов

На случай, если у вас нет локального HTTP-сервера для тестирования, я написал инструкцию по настройке с использованием только Python ⩾ 3.9 (без внешних библиотек) и поместил ее в файл *20-executors/getflags/README.adoc* (*https://github.com/fluentpython/example-code-2e/tree/master/20-executors/getflags*) в каталоге репозитория *fluentpython/examplecode-2e*. В двух словах *README.adoc* описывает, как использовать:

`python3 -m http.server`

Сервер `LOCAL`, прослушивающий порт 8000.

`python3 slow_server.py`

Сервер `DELAY`, прослушивающий порт 8001 и добавляющий случайную задержку 0 от 0,5 с до 5 с перед каждым ответом.

`python3 slow_server.py 8002 --error-rate .25`

Сервер `ERROR`, прослушивающий порт 8002, который помимо случайной задержки с вероятностью 25 % возвращает ответ «418 I'm a teapot», означающий ошибку.

По умолчанию каждый `flags2`-скрипт загружает флаги 20 стран с самым большим населением с локального сервера (`http://localhost:8000/flags`), открывая определенное количество соединений по умолчанию, для каждого скрипта свое. В примере 20.11 показан результат прогона скрипта *flags2_sequential.py*, когда все параметры заданы по умолчанию. Чтобы его выполнить, нужен локальный сервер, настроенный, как описано в примечании «Будьте осторожны при тестировании параллельных клиентов» выше.

Пример 20.11. Прогон flags2_sequential.py с параметрами по умолчанию: сервер LOCAL, флаги 20 самых густонаселенных стран, 1 соединение

```
$ python3 flags2_sequential.py
LOCAL site: http://localhost:8001/flags
Searching for 20 flags: from BD to VN
1 concurrent connection will be used.
--------------------
20 flags downloaded.
Elapsed time: 0.10s
```

Задать набор загружаемых флагов можно несколькими способами. В примере 20.12 показано, как загрузить флаги всех стран, коды которых начинаются с букв A, B, C.

Пример 20.12. Прогон flags2_threadpool.py для загрузки флагов всех стран, коды которых начинаются с букв A, B, C, с сервера DELAY

```
$ python3 flags2_threadpool.py -s DELAY a b c
DELAY site: http://localhost:8002/flags
Searching for 78 flags: from AA to CZ
30 concurrent connections will be used.
--------------------
43 flags downloaded.
35 not found.
Elapsed time: 1.72s
```

Независимо от способа задания кодов стран количество загружаемых флагов можно ограничить с помощью параметра `-l/--limit`. В примере 20.13 показано, как выполнить ровно 100 запросов с помощью комбинации параметра `-a`, запрашивающего все флаги, и параметра `-l 100`.

Пример 20.13. Прогон flags2_asyncio.py с загрузкой 100 флагов (`-al 100`) с сервера ERROR, 100 одновременных соединений (`-m 100`)

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8003/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
--------------------
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```

Так выглядит пользовательский интерфейс `flags2`-примеров. Теперь познакомимся с их реализацией.

Обработка ошибок во flags2-примерах

Общая стратегия обработки ошибок HTTP заключается в том, что ошибки 404 (Не найдено) обрабатываются функцией, отвечающей за загрузку одного файла (`download_one`), а все остальные исключения распространяются наружу и обрабатываются функцией `download_many` или сопрограммой `supervisor` (в примере `asyncio`).

И на этот раз начнем с рассмотрения последовательного кода, за выполнением которого легко проследить; многие функции из него будут использоваться и в многопоточном скрипте. В примере 20.14 показаны функции, которые, собственно, и выполняют загрузку в скриптах *flags2_sequential.py* и *flags2_threadpool.py*.

Пример 20.14. flags2_sequential.py: базовые функции, отвечающие за загрузку, обе используются также в скрипте flags2_threadpool.py

```
from collections import Counter
from http import HTTPStatus

import httpx
import tqdm # type: ignore ❶

from flags2_common import main, save_flag, DownloadStatus ❷

DEFAULT_CONCUR_REQ = 1
MAX_CONCUR_REQ = 1

def get_flag(base_url: str, cc: str) -> bytes:
    url = f'{base_url}/{cc}/{cc}.gif'.lower()
    resp = httpx.get(url, timeout=3.1, follow_redirects=True)
    resp.raise_for_status() ❸
    return resp.content

def download_one(cc: str, base_url: str, verbose: bool = False) -> DownloadStatus:
    try:
        image = get_flag(base_url, cc)
```

```
    except httpx.HTTPStatusError as exc: ❹
        res = exc.response
        if res.status_code == HTTPStatus.NOT_FOUND:
            status = DownloadStatus.NOT_FOUND ❺
            msg = f'not found: {res.url}'
        else:
            raise ❻
    else:
        save_flag(image, f'{cc}.gif')
        status = DownloadStatus.OK
        msg = 'OK'

    if verbose: ❼
        print(cc, msg)

    return status
```

❶ Импортировать библиотеку `tqdm` для отображения индикатора хода выполнения и сказать Муру, что проверять ее не надо[1].
❷ Импортировать две функции и перечисление `Enum` из модуля `flags2_common`.
❸ Возбуждает исключение `HTTPStatusError`, если код состояния HTTP не принадлежит диапазону `range(200, 300)`.
❹ Функция `download_one` перехватывает исключение `HTTPStatusError`, чтобы обработать ошибку с кодом 404 и только ее…
❺ …установив локальное состояние `status` равным `DownloadStatus.NOT_FOUND`; `DownloadStatus` – перечисление, импортированное из *flags2_common.py*.
❻ Любое другое исключение типа `HTTPStatusError` возбуждается повторно и распространяется в вызывающую программу.
❼ Если задан параметр `-v/--verbose`, то отображается сообщение, содержащее код страны и состояние; именно так мы видим индикатор хода выполнения в режиме вывода подробной информации.

В примере 20.15 приведена последовательная версия функции `download_many`. Ее код прямолинеен, но его стоит изучить хотя бы для сравнения с конкурентными версиями. Обратите внимание на индикацию хода выполнения, обработку ошибок и подсчет количества загрузок с разным исходом.

Пример 20.15. flags2_sequential.py: последовательная реализация `download_many`

```
def download_many(cc_list: list[str],
                  base_url: str,
                  verbose: bool,
                  _unused_concur_req: int) -> Counter[DownloadStatus]:
    counter: Counter[DownloadStatus] = Counter() ❶
    cc_iter = sorted(cc_list) ❷
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter) ❸
    for cc in cc_iter:
        try:
            status = download_one(cc, base_url, verbose) ❹
```

[1] По состоянию на сентябрь 2021 года в *tdqm* не было аннотаций типов. Это нормально. Мир из-за этого не рухнет. Спасибо Гвидо за факультативную типизацию!

```
            except httpx.HTTPStatusError as exc: ❺
                error_msg = 'HTTP error {resp.status_code} - {resp.reason_phrase}'
                error_msg = error_msg.format(resp=exc.response)
            except httpx.RequestError as exc: ❻
                error_msg = f'{exc} {type(exc)}'.strip()
            except KeyboardInterrupt: ❼
                break
            else: ❽
                error_msg = ''

            if error_msg:
                status = DownloadStatus.ERROR ❾
            counter[status] += 1 ❿
            if verbose and error_msg: ⓫
                print(f'{cc} error: {error_msg}')

    return counter ⓬
```

❶ Этот объект `Counter` подсчитывает количество загрузок с разными исходами: `DownloadStatus.OK`, `DownloadStatus.NOT_FOUND`, `DownloadStatus.ERROR`.

❷ В `cc_iter` хранится отсортированный по алфавиту список кодов стран, полученных в виде аргументов.

❸ Если не задан режим подробной информации, то `cc_iter` передается функции `tqdm`, которая возвращает итератор `cc_iter`, отдающий элементы, и одновременно отображает анимированный индикатор хода выполнения.

❹ Последовательные обращения к `download_one`.

❺ Относящиеся к HTTP исключения, возбужденные функцией `get_flag` и не обработанные в `download_one`, обрабатываются здесь.

❻ Прочие относящиеся к сети исключения обрабатываются здесь. Все остальные исключения аварийно завершают скрипт, потому что в функции `flags2_common.main`, из которой вызывается `download_many`, нет блока `try/except`.

❼ Выйти из цикла, если пользователь нажал **Ctrl-C**.

❽ Если исключение не вышло за пределы `download_one`, очистить сообщение об ошибке.

❾ Если произошла ошибка, устанавливаем соответствующее значение `status`.

❿ Увеличиваем счетчик для этого значения `status`.

⓫ При работе в режиме подробной информации отображаем сообщение об ошибке для текущего кода страны, если таковое имеется.

⓬ Возвращаем `counter`, чтобы функция `main` могла вывести финальный отчет.

Теперь рассмотрим переработанный пример с пулом потоков, *flags2_threadpool.py*.

Использование futures.as_completed

Чтобы включить индикатор хода выполнения и обработку ошибок, мы используем в скрипте *flags2_threadpool.py* класс `futures.ThreadPoolExecutor` совместно с уже встречавшейся функцией `futures.as_completed`. В примере 20.16 приведен полный код *flags2_threadpool.py*. Заново реализована только функция `download_many`; все остальные функции заимствованы из модулей `flags2_common` и `flags2_sequential`.

Пример 20.16. flags2_threadpool.py: полный исходный код

```
from collections import Counter
from concurrent.futures import ThreadPoolExecutor, as_completed

import httpx
import tqdm # type: ignore

from flags2_common import main, DownloadStatus
from flags2_sequential import download_one ❶

DEFAULT_CONCUR_REQ = 30 ❷
MAX_CONCUR_REQ = 1000 ❸

def download_many(cc_list: list[str],
                  base_url: str,
                  verbose: bool,
                  concur_req: int) -> Counter[DownloadStatus]:
    counter: Counter[DownloadStatus] = Counter()
    with ThreadPoolExecutor(max_workers=concur_req) as executor: ❹
        to_do_map = {} ❺
        for cc in sorted(cc_list): ❻
            future = executor.submit(download_one, cc,
                                     base_url, verbose) ❼
            to_do_map[future] = cc ❽
        done_iter = as_completed(to_do_map) ❾
        if not verbose:
            done_iter = tqdm.tqdm(done_iter, total=len(cc_list)) ❿
        for future in done_iter: ⓫
            try:
                status = future.result() ⓬
            except httpx.HTTPStatusError as exc: ⓭
                error_msg = 'HTTP error {resp.status_code} - {resp.reason_phrase}'
                error_msg = error_msg.format(resp=exc.response)
            except httpx.RequestError as exc:
                error_msg = f'{exc} {type(exc)}'.strip()
            except KeyboardInterrupt:
                break
            else:
                error_msg = ''

            if error_msg:
                status = DownloadStatus.ERROR
            counter[status] += 1
            if verbose and error_msg:
                cc = to_do_map[future] ⓮
                print(f'{cc} error: {error_msg}')

    return counter

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

❶ Повторно использовать функцию `donwload_one` из `flags2_sequential` (пример 20.14).

❷ Если в командной строке не задан параметр `-m/--max_req`, то принимаем такое максимальное число конкурентных запросов, оно и станет размером пула потоков. Фактическое число потоков может быть меньше, если загружается меньше флагов.

❸ `MAX_CONCUR_REQ` – максимальное число конкурентных запросов независимо от числа загружаемых флагов и от значения параметра `-m/--max_req`; это мера предосторожности, позволяющая избежать запуска слишком большого числа потоков и чрезмерного потребления памяти.

❹ Создать объект `executor` с параметром `max_workers`, равным величине `concur_req`, которую функция `main` вычисляет как минимум из `MAX_CONCUR_REQ`, длины списка `cc_list`, и значения параметра командной строки `-m/--max_req`. Это позволяет избежать создания большего числа потоков, чем необходимо.

❺ Этот словарь отображает каждый экземпляр `Future` – представляющий одну загрузку – на соответствующий код страны для показа в сообщении об ошибке.

❻ Обходим список кодов стран в алфавитном порядке. Порядок результатов зависит прежде всего от времени получения HTTP-ответа, но если размер пула (определяемый величиной `concur_req`) гораздо меньше `len(cc_list)`, то может оказаться, что результаты возвращаются по алфавиту.

❼ Каждое обращение к `executor.submit` планирует выполнение одного вызываемого объекта и возвращает экземпляр `Future`. Первый аргумент – сам вызываемый объект, остальные – передаваемые ему аргументы.

❽ Сохранить `future` и код страны в словаре.

❾ Функция `futures.as_completed` возвращает итератор, который отдает будущие объекты по мере их завершения.

❿ Если не установлен режим подробной информации, то обертываем результат `as_completed` функцией `tqdm`, которая отображает индикатор хода выполнения; поскольку у `done_iter` нет метода `len`, мы должны сообщить `tqdm` ожидаемое количество элементов в виде аргумента `total=`, чтобы `tqdm` могла оценить объем оставшейся работы.

⓫ Обойти будущие объекты по мере их завершения.

⓬ Вызов метода `result` будущего объекта возвращает значение, полученное от вызываемого объекта, или возбуждает исключение, которое было перехвачено во время выполнения объекта. Этот метод может блокировать программу в ожидании разрешения ситуации, но не в данном примере, потому что `as_completed` возвращает только уже завершенные будущие объекты.

⓭ Обработать потенциальные исключения; оставшаяся часть функции отличается от кода последовательной версии `download_many` (пример 20.15) только в месте следующей выноски.

⓮ Чтобы предоставить контекст для сообщения об ошибке, извлекаем код страны из словаря `to_do_map`, используя в качестве ключа текущий объект `future`. В последовательной версии это было необязательно, потому что мы обходили список кодов стран, так что текущий `cc` всегда был под рукой; здесь же мы обходим будущие объекты.

В примере 20.16 используется идиома, очень полезная при работе с функцией `futures.as_completed`: построить словарь, ставящий в соответствие каждому будущему объекту данные, которые можно будет использовать по завершении этого объекта. В данном случае словарь `to_do_map` сопоставляет с будущим объектом соответствующий ему код страны. Это упрощает последующую обработку будущих объектов, несмотря на то что завершаться они могут не по порядку.

Потоки Python отлично приспособлены к приложениям с большим объемом ввода-вывода, а благодаря пакету `concurrent.futures` их использование в ряде случаев оказывается сравнительно простым. С помощью класса `ProcessPoolExecutor` мы также можем решать счетные задачи на нескольких ядрах – если вычисления «естественно параллельны» (*https://en.wikipedia.org/wiki/Embarrassingly_parallel*). На этом мы завершаем введение в пакет `concurrent.futures`.

Резюме

В начале этой главы мы сравнили два параллельных HTTP-клиента с последовательным и убедились в том, что распараллеливание дает значительный выигрыш в производительности.

Изучив первый пример, основанный на пакете `concurrent.futures`, мы решили поближе познакомиться с будущими объектами – экземплярами класса `concurrent.futures.Future` или `asyncio.Future`, – уделив особое внимание общим чертам этих классов (различия между ними будут рассмотрены в главе 21). Мы видели, как создавать будущие объекты методом `Executor.submit` и как обходить завершенные объекты с помощью функции `concurrent.futures.as_completed`.

Затем обсудили запуск нескольких процессов с помощью класса `concurrent.futures.ProcessPoolExecutor`, что позволяет обойти ограничение GIL и, задействовав несколько процессорных ядер, упростить программу проверки чисел на простоту из главы 19.

В следующем разделе мы изучили, как работает класс `concurrent.futures.ThreadPoolExecutor` на учебном примере, где запускались задачи, которые просто спали несколько секунд, ничего не делая, а затем печатали свое состояние и временную метку.

Далее мы вернулись к примерам загрузки изображений флагов. Чтобы добавить в них индикатор хода выполнения и корректную обработку ошибок, нам пришлось углубиться в детали генераторной функции `future.as_completed`, и в результате мы открыли для себя общий прием: сохранение будущих объектов в словаре вместе с дополнительной информацией в момент передачи исполнителю и использование этой информации впоследствии – когда итератор `as_completed` отдает завершенный будущий объект.

Дополнительная литература

Автором пакета `concurrent.futures` является Брайан Куинлан, который презентовал его в блестящем докладе под названием «Будущее уже близко!» (*https://pyvideo.org/pycon-au-2010/pyconau-2010-the-future-is-soon.html*) на конференции

PyCon Australia 2010. Доклад Куинлана не сопровождается слайдами; он демонстрирует возможности библиотеки, вводя код прямо в оболочке Python. В качестве пояснительного примера на презентации был показан короткий ролик, созданный автором веб-комикса и программистом Рэнделлом Манро, в котором он непреднамеренно организовал DoS-атаку на сайт Google Maps, чтобы построить цветную карту времени поездок на автомобиле в своем городе. Формальное введение в библиотеку содержится в документе PEP 3148 «`futures` – execute computations asynchronously» (*https://www.python.org/dev/peps/pep-3148/*). В нем Куинлан пишет, что на библиотеку `concurrent.futures` «большое влияние оказал пакет `java.util.concurrent` для Java».

Дополнительные ресурсы, относящиеся к пакету `concurrent.futures`, рассматриваются в главе 19. Все ссылки, касающиеся пакетов `threading` и `multiprocessing`, приведенные в разделе «Конкурентность с применением потоков и процессов» главы 19, относятся к пакету `concurrent.futures`.

Поговорим

Держаться подальше от потоков

> Конкурентность – один из самых трудных вопросов информатики (лучше держаться от него подальше).
>
> – Дэвид Бизли, преподаватель Python и безумный ученый[1]

Я согласен с, казалось бы, противоречащими друг другу высказываниями Дэвида Бизли (см. выше) и Мишеля Симионато (взято в качестве эпиграфа к этой главе).

Я прослушал в университете курс по конкурентности. Там мы занимались только программированием потоков POSIX (*https://en.wikipedia.org/wiki/Pthreads*). И я пришел к выводу, что управлять потоками и блокировками самостоятельно я хочу ничуть не больше, чем заниматься выделением и освобождением памяти. Такие вещи лучше оставить системным программистам, которые знают, как это делать, любят с этим возиться и располагают временем, чтобы сделать все правильно, – по крайней мере, я надеюсь на это. Мне же платят за разработку приложений, а не операционных систем. Мне ни к чему точный контроль над потоками, блокировками, `malloc` и `free` – см. статью «Динамическое распределение памяти в C» (*https://en.wikipedia.org/wiki/C_dynamic_memory_allocation*).

Потому-то я и считаю пакет `concurrent.futures` выдающимся достижением: в нем потоки, процессы и очереди рассматриваются как элементы инфраструктуры, а не как объекты, с которыми нужно работать напрямую. Конечно, этот пакет предназначен для сравнительно простых задач, которые принято называть естественно параллельными. Но это довольно большая часть всего множества проблем распараллеливания, с которыми приходится сталкиваться при разработке приложений – в противоположность операционным системам или серверам баз данных, – о чем и говорит Симионато.

[1] Слайд 9 из пособия «A Curious Course on Coroutines and Concurrency» (*http://www.dabeaz.com/coroutines/*), представленного на конференции PyCon 2009.

Для задач, не являющихся «естественно параллельными», потоки и блокировки – тоже не решение. На уровне ОС потоки никогда не исчезнут, но во всех языках программирования, которые мне кажутся интересными, за последние несколько лет появились более удобные высокоуровневые абстракции конкурентности, как показывает книга Пола Бутчера «Seven Concurrency Models in Seven Weeks» (*https://pragprog.com/titles/pb7con/seven-concurrency-models-in-seven-weeks/*). К числу таких языков относятся Go, Elixir и Clojure. Erlang – язык, на котором написан Elixir, – блестящий пример языка, в который уже на этапе проектирования был заложен параллелизм. Мне, впрочем, он не нравится из-за уродливого синтаксиса. Это меня Python избаловал.

Хосе Валим, хорошо известный как один из авторов ядра Ruby on Rails, спроектировал язык Elixir, наделив его приятным современным синтаксисом. Подобно Lisp и Clojure, в Elixir реализованы синтаксические макросы. Но это палка о двух концах. Синтаксические макросы позволяют строить мощные предметно-ориентированные языки (DSL), но чрезмерное изобилие подъязыков может привести к несовместимым кодовым базам и фрагментации сообщества. Lisp утонул в макросах, каждый поставщик Lisp предлагает свой собственный сокровенный диалект. Результатом стандартизации на основе Common Lisp стал язык, разбухший от функциональных возможностей. Надеюсь, Хосе Валим не даст сообществу Elixir пойти по тому же пути. Пока все выглядит неплохо. Оберткой баз данных и генератором запросов Ecto (*https://hexdocs.pm/ecto/getting-started.html*) пользоваться одно удовольствие: прекрасный пример использования макросов для создания гибкого, но при этом дружественного пользователю предметно-ориентированного языка (DSL) для взаимодействия с реляционными и нереляционными базами данных.

Как и Elixir, Go – современный язык со свежими идеями. Но в некоторых отношениях он по сравнению с Elixir консервативен. В Go нет макросов, а его синтаксис проще, чем в Python. Go не поддерживает ни наследование, ни перегрузку операторов и предлагает меньше средств для метапрограммирования, чем Python. Эти ограничения рассматриваются как достоинства. Они позволяют обеспечить более предсказуемые поведение и производительность. И это большой плюс в тех особо ответственных задачах с высоким уровнем параллелизма, где Go рассчитывает заменить C++, Java и Python.

Хотя Elixir и Go – прямые конкуренты на поле конкурентности, их философия рассчитана на разную аудиторию. Скорее всего, обоим языкам уготована счастливая судьба. Но история учит, что более консервативные языки программирования привлекают больше последователей.

Глава 21

Асинхронное программирование

> Проблема всех обычных подходов к асинхронному программированию в том, что они предлагают все или ничего. Вам придется переписать весь код, чтобы нигде ничего не блокировалось, иначе вы просто зря потратите время.
>
> – Альваро Видела и Джейсон Дж. У. Уильямс, «RabbitMQ in Action»[1]

В этой главе рассматриваются три большие, тесно связанные между собой темы:

- конструкции Python `async def`, `await`, `async with` и `async for`;
- объекты, поддерживающие эти конструкции: платформенные сопрограммы и асинхронные варианты контекстных менеджеров, итерируемых объектов, генераторов и включений;
- *asyncio* и другие асинхронные библиотеки.

Материал этой главы основан на идеях итерируемых объектов и генераторов (глава 17, в частности раздел «Классические сопрограммы»), контекстных менеджеров (глава 18) и общих концепциях конкурентного программирования (глава 19).

Мы будем изучать конкурентные HTTP-клиенты, подобные рассмотренным в главе 20, но перепишем их с применением платформенных сопрограмм и асинхронных контекстных менеджеров. Для этой цели будем пользоваться той же библиотекой *HTTPX*, что и раньше, но на этот раз ее асинхронным API. Мы также увидим, как можно избежать блокирования цикла событий, делегируя медленные операции исполнителю в отдельном потоке или процессе.

После примеров HTTP-клиентов мы рассмотрим два простых асинхронных серверных приложения, одно из которых основано на каркасе *FastAPI*, быстро набирающем популярность. Затем обратимся к другим языковым конструкциям, связанным с ключевыми словами `async` и `await`: асинхронным генераторным функциям, асинхронным включениям и асинхронным генераторным выражениям. Чтобы подчеркнуть, что эти языковые средства не привязаны к библиотеке *asyncio*, мы перепишем один пример с использованием *Curio* – элегантного новаторского асинхронного каркаса, написанного Дэвидом Бизли.

---

[1] *Videla & Williams.* RabbitMQ in Action (Manning); глава 4 «Solving Problems with Rabbit: coding and patterns. С. 61.

И в завершение этой главы я написал короткий раздел о преимуществах и недостатках асинхронного программирования.

Так что впереди у нас много работы. Место нашлось только для простых примеров, но они иллюстрируют самые важные аспекты каждой идеи.

Документация по библиотеке *asyncio* (*https://docs.python.org/3/library/asyncio.html*) стала гораздо лучше, после того ее реорганизовал Юрий Селиванов[1], отделив немногие функции, полезные разработчикам приложений, от низкоуровневого API, предназначенного для создателей пакетов, например веб-каркасов и драйверов баз данных.

Если вас интересует целая книга, посвященная *asyncio*, рекомендую Caleb Hattingh «Using Asyncio in Python» (O'Reilly) (*https://www.oreilly.com/library/view/using-asyncio-in/9781492075325/*). Замечание: Калеб – один из технических рецензентов этой книги.

Что нового в этой главе

Когда я работал над первым изданием этой книги, библиотека *asyncio* еще не устоялась, а ключевых слов `async` и `await` не было вовсе. Поэтому мне пришлось обновить все примеры в данной главе. Я написал также новые примеры: скрипты проверки доменных имен, веб-службу на основе *FastAPI* и эксперименты с новым асинхронным режимом консоли Python.

В новых разделах рассматриваются языковые средства, которых в то время еще не было: платформенные сопрограммы, `async with`, `async for` и объекты, поддерживающие эти конструкции.

Идеи, изложенные в разделе «Как работает асинхронный код и как он не работает», отражают выученные тяжким трудом уроки, которые, на мой взгляд, должен усвоить каждый, кто пользуется асинхронным программированием. Они могут избавить вас от кучи неприятностей – и не важно, пишете вы на Python или на Node.js.

Наконец, я убрал несколько абзацев, посвященных классу `asyncio.Futures`, который теперь считается низкоуровневой частью API *asyncio*.

Несколько определений

В начале раздела «Классические сопрограммы» главы 17 мы видели, что начиная с версии Python 3.5 предлагается три вида сопрограмм:

Платформенная сопрограмма

Функция, определенная с помощью конструкции `async def`. Мы можем делегировать работу от одной платформенной сопрограммы другой, воспользовавшись ключевым словом `await`, по аналогии с тем, как классические сопрограммы уступают управление с помощью предложения `yield from`. Предложение `async def` всегда определяет платформенную сопрограмму, даже

[1] Селиванов реализовал `async/await` в Python и написал связанные с этим документы PEP 492, 525 и 530.

если в ее теле не встречается ключевое слово `await`. Слово `await` нельзя использовать вне платформенной сопрограммы[1].

Классическая сопрограмма

Генераторная функция, которая потребляет данные, отправленные ей с помощью вызовов `my_coro.send(data)`, и читает эти данные, используя `yield` в выражении. Классическая сопрограмма может делегировать работу другой классической сопрограмме с помощью предложения `yield from`. Классические сопрограммы не приводятся в действие словом `await` и более не поддерживаются библиотекой *asyncio*.

Генераторные сопрограммы

Генераторная функция, снабженная декоратором `@types.coroutine`, включенным в Python 3.5. Этот декоратор делает генератор совместимым с новым ключевым словом `await`.

В этой главе нас будут интересовать платформенные сопрограммы и *асинхронные генераторы*:

Асинхронный генератор

Генераторная функция, определенная с помощью конструкции `async def` и содержащая в теле `yield`. Она возвращает асинхронный объект-генератор, предоставляющий метод `__anext__` для асинхронного получения следующего элемента.

@asyncio.coroutine не имеет будущего[2]

Декоратор `@asyncio.coroutine` для классических и генераторных сопрограмм был объявлен нерекомендуемым в версии 3.8, а в версии Python 3.11 его планируется исключить из языка, как написано в сообщении 43216 (*https://bugs.python.org/issue43216*). Напротив, декоратор `@types.coroutine` должен остаться, если верить сообщению 36921 (*https://bugs.python.org/issue36921*). Он больше не поддерживается *asyncio*, но используется в низкоуровневом коде асинхронных каркасов *Curio* и *Trio*.

Пример использования asyncio: проверка доменных имен

Допустим, вы собираетесь начать новый блог, посвященный Python, и планируете зарегистрировать домен, содержащий какое-нибудь ключевое слово Python и имеющий суффикс *.DEV*, например *AWAIT.DEV*. В примере 21.1 показан скрипт, в котором *asyncio* используется для конкурентной проверки нескольких доменных имен. Вот его выход:

```
$ python3 blogdom.py
  with.dev
+ elif.dev
```

[1] Из этого правила есть одно исключение: если Python запущен с флагом `-m asyncio`, то `await` можно использовать прямо после приглашения `>>>` для вызова платформенной сопрограммы. Это объяснено в разделе «Эксперименты с асинхронной консолью Python».

[2] Извините, не мог противиться искушению.

```
+ def.dev
  from.dev
  else.dev
  or.dev
  if.dev
  del.dev
+ as.dev
  none.dev
  pass.dev
  true.dev
+ in.dev
+ for.dev
+ is.dev
+ and.dev
+ try.dev
+ not.dev
```

Обратите внимание, что доменные имена не упорядочены. Запустив скрипт, вы увидите, что они отображаются один за другим с переменной задержкой. Знак `+` означает, что машина смогла разрешить доменное имя. В противном случае доменное имя не существует и, возможно, свободно[1].

В скрипте *blogdom.py* опрос DNS производится с помощью объектов платформенных сопрограмм. Поскольку асинхронные операции чередуются, время, необходимое для проверки всех 18 доменов, гораздо меньше, чем если бы они проверялись последовательно. На самом деле полное время практически совпадает со временем получения самого медленного ответа от DNS, а не равно сумме времен всех ответов.

Пример 21.1. blogdom.py: поиск доменного имени для блога о Python

```
#!/usr/bin/env python3
import asyncio
import socket
from keyword import kwlist

MAX_KEYWORD_LEN = 4 ❶

async def probe(domain: str) -> tuple[str, bool]: ❷
    loop = asyncio.get_running_loop() ❸
    try:
        await loop.getaddrinfo(domain, None) ❹
    except socket.gaierror:
        return (domain, False)
    return (domain, True)

async def main() -> None: ❺
    names = (kw for kw in kwlist if len(kw) <= MAX_KEYWORD_LEN) ❻
    domains = (f'{name}.dev'.lower() for name in names) ❼
    coros = [probe(domain) for domain in domains] ❽
    for coro in asyncio.as_completed(coros): ❾
        domain, found = await coro ❿
        mark = '+' if found else ' '
        print(f'{mark} {domain}')
```

[1] Когда я это писал, домен `true.dev` предлагался за 360 долларов США в год. Я видел, что домен `for.dev` зарегистрирован, но DNS-запись не была сконфигурирована.

```
if __name__ == '__main__':
    asyncio.run(main()) ⓫
```

❶ Задать максимальную длину ключевого слова в доменном имени, поскольку чем оно короче, тем лучше.

❷ Функция `probe` возвращает кортеж, содержащий доменное имя и булево значение; `True` означает, что имя успешно разрешено. Возврат доменного имени упрощает отображение результатов.

❸ Получить ссылку на цикл событий `asyncio` для будущего использования.

❹ Метод-сопрограмма `loop.getaddrinfo(…)` (*https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.loop.getaddrinfo*) возвращает 5-кортеж параметров (*https://docs.python.org/3/library/socket.html#socket.getaddrinfo*) для подключения к указанному адресу через сокет. В этом примере нам результат не нужен. Если мы получили кортеж, значит, имя разрешено, в противном случае – нет.

❺ `main` должна быть сопрограммой, чтобы в ней можно было использовать `await`.

❻ Генератор, отдающий ключевые слова Python длиной не более `MAX_KEYWORD_LEN`.

❼ Генератор, отдающий доменные имена с суффиксом `.dev`.

❽ Построить список объектов сопрограмм, вызывая сопрограмму `probe` с каждым аргументом `domain`.

☒ `asyncio.as_completed` – генератор, отдающий переданные ему сопрограммы в порядке их завершения, а не в порядке подачи. Он похож на функцию `futures.as_completed`, которую мы видели в примере 20.4.

❿ В этот момент мы знаем, что сопрограмма завершилась, потому что так работает `as_completed`. Поэтому выражение `await` не заблокирует выполнение, но оно все равно необходимо, чтобы получить результат от `coro`. Если `coro` возбуждала необработанное исключение, то оно будет заново возбуждено в этой точке.

⓫ `asyncio.run` запускает цикл событий и возвращает управление только после выхода из него. Это типичный паттерн для скриптов, в которых используется `asyncio`: реализовать `main` как сопрограмму и выполнить ее внутри блока `if __name__ == '__main__':`.

Функция `asyncio.get_running_loop` была добавлена в версии Python 3.7 для использования внутри сопрограмм, как показано в `probe`. Если работающего цикла нет, то она возбуждает исключение `RuntimeError`. Ее реализация проще и быстрее, чем функции `asyncio.get_event_loop`, которая может при необходимости запустить цикл событий. Начиная с версии Python 3.10 `asyncio.get_event_loop` объявлена нерекомендуемой (*https://docs.python.org/3.10/library/asyncio-eventloop.html#asyncio.get_event_loop*) и в конечном итоге станет псевдонимом `asyncio.get_running_loop`.

Предложенный Гвидо способ чтения асинхронного кода

В *asyncio* много новых концепций, которые предстоит переварить, но за общей логикой примера 21.1 будет легко следить, если воспользоваться приемом, предложенным самим Гвидо ван Россумом: прищуриться и сделать вид, что ключевых слов `async` и `await` нет. Тогда вы поймете, что сопрограммы читаются, как старые добрые последовательные функции.

Например, представьте, что тело сопрограммы…

```
async def probe(domain: str) -> tuple[str, bool]:
    loop = asyncio.get_running_loop()
    try:
        await loop.getaddrinfo(domain, None)
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

…работает, как следующая функция, с тем отличием, что волшебным образом никогда не блокирует выполнение программы:

```
def probe(domain: str) -> tuple[str, bool]: # no async
    loop = asyncio.get_running_loop()
    try:
        loop.getaddrinfo(domain, None) # no await
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

Конструкция `await loop.getaddrinfo(...)` позволяет избежать блокирования, потому что `await` приостанавливает текущий объект сопрограммы. Например, во время выполнения сопрограммы `probe('if.dev')` создается новый объект сопрограммы с помощью вызова `getaddrinfo('if.dev', None)`. Его ожидание запускает низкоуровневый запрос `addrinfo` и уступает управление циклу событий, а не приостановленной сопрограмме `probe('if.dev')`. Затем цикл событий может передать управление другим ожидающим объектам сопрограмм, например `probe('or.dev')`.

Когда цикл событий получит ответ на запрос `getaddrinfo('if.dev', None)`, этот объект сопрограммы возобновляется и возвращает управление `probe('if.dev')`, которая была приостановлена в `await`, а теперь может обработать возможное исключение и вернуть кортеж с результатами.

До сих пор мы видели только, как `asyncio.as_completed` и `await` применяются к сопрограммам. Но они могут обработать любой *допускающий ожидание* объект. Это понятие объясняется ниже.

Новая концепция: объекты, допускающие ожидание

Ключевое слово `for` работает с *итерируемыми объектами*. А ключевое слово `await` – с объектами, *допускающими ожидание*.

Конечный пользователь *asyncio* постоянно сталкивается со следующими объектами, допускающими ожидание:

- *объект платформенной сопрограммы*, который мы получаем в результаты вызова *функции платформенной сопрограммы*;
- `asyncio.Task`, который мы обычно получаем, передав объект сопрограммы функции `asyncio.create_task()`.

Однако пользовательский код не всегда должен ожидать `Task` с помощью `await`. Мы используем `asyncio.create_task(one_coro())`, чтобы запланировать конкурентное выполнение `one_coro` и не ждать, пока она вернется. Именно так мы поступили с сопрограммой `spinner` в скрипте *spinner_async.py* (пример 19.4).

Если вы не собираетесь отменять задачу или ждать ее завершения, то и не нужно хранить объект `Task`, возвращенный функцией `create_task`. Достаточно просто создать задачу, чтобы запланировать выполнение сопрограммы.

С другой стороны, мы используем `await other_coro()`, чтобы выполнить `other_coro` немедленно и дождаться ее завершения, потому что для продолжения работы нужен ее результат. В скрипте *spinner_async.py* в сопрограмме `supervisor` мы писали `res = await slow()`, чтобы выполнить `slow` и получить ее результат.

При реализации асинхронных библиотек или дополнений самой библиотеки *asyncio* иногда приходится иметь дело со следующими низкоуровневыми объектами, допускающими ожидание:

- объект, имеющий метод `__await__`, который возвращает итератор; например, экземпляр `asyncio.Future` (`asyncio.Task` является подклассом `asyncio.Future`);
- объекты, написанные на других языках с применением Python/C API, имеющие функцию `tp_as_async.am_await`, возвращающую итератор (аналогична методу `__await__`).

В существующих кодовых базах может встречаться еще один вид объекта, допускающего ожидание: *объекты генераторных сопрограмм*. Но их скоро собираются объявить нерекомендуемыми.

В документе PEP 492 утверждается (*https://peps.python.org/pep-0492/#await-expression*), что выражение `await` «использует реализацию `yield from` с дополнительным шагом проверки аргумента» и что «`await` принимает только объект, допускающий ожидание». В PEP эта реализация не объясняется подробно, но дается отсылка к документу PEP 380 (*https://peps.python.org/pep-0380*), в котором впервые было описано предложение `yield from`. Я поместил детальное объяснение в раздел «Семантика yield from» (*https://www.fluentpython.com/extra/classic-coroutines/#yield_from_meaning_sec*) статьи «Классические сопрограммы» на сайте *fluentpython.com*.

Теперь перейдем к изучению версии скрипта, загружающего изображения флагов, написанной с применением *asyncio*.

Загрузка файлов с помощью asyncio и HTTPX

Скрипт *flags_asyncio.py* загружает фиксированный набор 20 флагов с сайта *fluentpython.com*. Впервые мы упоминали о нем в разделе «Конкурентная загрузка из веба» главы 20, а теперь изучим более подробно, применив только что полученные знания.

В версии Python 3.10 *asyncio* непосредственно поддерживает только протоколы TCP и UDP, и в стандартной библиотеке нет пакетов с асинхронным HTTP-клиентом или сервером. Во всех примерах HTTP-клиентов я буду пользоваться библиотекой *HTTPX* (*https://www.python-httpx.org/*).

Мы будем изучать скрипт *flags_asyncio.py* снизу вверх, т. е. сначала рассмотрим функции, которые подготавливают действие (пример 21.2).

Чтобы код было проще читать, в скрипте *flags_asyncio.py* нет обработки ошибок. Знакомясь с `async/await`, полезно сначала сосредоточить внимание на «успешном пути», чтобы понять, как организованы регулярные функции и сопрограммы. Начиная с раздела «Улучшение асинхронного загрузчика» примеры включают обработку ошибок и дополнительные возможности.

Примеры скриптов *flags_.py* из этой главы и главы 20 разделяют общий код и данные, поэтому я поместил их в каталог *example-code-2e/20-executors/getflags*.

Пример 21.2. flags_asyncio.py: функции подготовки

```
def download_many(cc_list: list[str]) -> int: ❶
    return asyncio.run(supervisor(cc_list)) ❷

async def supervisor(cc_list: list[str]) -> int:
    async with AsyncClient() as client: ❸
        to_do = [download_one(client, cc)
                 for cc in sorted(cc_list)] ❹
        res = await asyncio.gather(*to_do) ❺

    return len(res) ❻

if __name__ == '__main__':
    main(download_many)
```

❶ Это должна быть обычная функция, а не сопрограмма, чтобы ее можно было передать и вызвать из функции `main`, находящейся в модуле *flags.py* (пример 20.2).

❷ Выполнять цикл событий, приводящий в действие объект сопрограммы `supervisor(cc_list)`, пока тот не вернет управление. Эта строка блокирует выполнение на все время работы цикла событий. Ее результатом является значение, возвращенное `supervisor`.

❸ Асинхронные операции HTTP-клиента в `httpx` – это методы класса `AsyncClient`, который также является асинхронным контекстным менеджером, т. е. контекстным менеджером с асинхронными методами инициализации и очистки (подробнее об этом см. в разделе «Асинхронные контекстные менеджеры»).

❹ Построить список объектов сопрограмм, вызвав сопрограмму `download_one` по разу для каждого флага.

❺ Ждать завершения сопрограммы `asyncio.gather`, которая принимает один или несколько допускающих ожидание аргументов, ждет их завершения, а затем возвращает список результатов заданных объектов в том порядке, в каком они подавались на вход.

❻ `supervisor` возвращает длину списка, возвращенного функцией `asyncio.gather`.

Теперь рассмотрим начало файла *flags_asyncio.py* (пример 21.3). Я реорганизовал сопрограммы, так чтобы их можно было читать в том порядке, в каком они запускаются циклом событий.

Пример 21.3. flags_asyncio.py: импорт и функции загрузки

```
import asyncio

from httpx import AsyncClient ❶

from flags import BASE_URL, save_flag, main ❷

async def download_one(client: AsyncClient, cc: str): ❸
    image = await get_flag(client, cc)
    save_flag(image, f'{cc}.gif')
    print(cc, end=' ', flush=True)
    return cc

async def get_flag(client: AsyncClient, cc: str) -> bytes: ❹
    url = f'{BASE_URL}/{cc}/{cc}.gif'.lower()
    resp = await client.get(url, timeout=6.1,
                                 follow_redirects=True) ❺
    return resp.read() ❻
```

❶ `httpx` нужно устанавливать – это не часть стандартной библиотеки.
❷ Повторно использовать код из *flags.py* (пример 20.2).
❸ `download_one` должна быть платформенной сопрограммой, чтобы она могла вызвать `await` для сопрограммы `get_flag`, которая выполняет HTTP-запрос. Затем она отображает загруженный флаг и сохраняет изображение.
❹ `get_flag` должна получить `AsyncClient`, чтобы сделать запрос.
❺ Метод `get` экземпляра `httpx.AsyncClient` возвращает объект `ClientResponse`, который заодно является асинхронным контекстным менеджером.
❻ Операции сетевого ввода-вывода реализованы в виде методов-сопрограмм, чтобы их можно было асинхронно вызывать из цикла событий `asyncio`.

Для повышения производительности вызов `save_flag` из `get_flag` следовало бы сделать асинхронным, чтобы не блокировать цикл событий. Однако в настоящее время *asyncio* не предоставляет асинхронного API файловой системы – в отличие от Node.js.

В разделе «Использование asyncio.as_completed в сочетании с потоком» ниже будет показано, как делегировать работу `save_flag` потоку.

Наш код делегирует работу сопрограммам `httpx` явно с помощью `await` или неявно с помощью специальных методов асинхронных контекстных менеджеров, например `AsyncClient` и `ClientResponse`. Последнее мы увидим в разделе «Асинхронные контекстные менеджеры» ниже.

Секрет платформенных сопрограмм: скромные генераторы

Ключевое различие между примерами классических сопрограмм в разделе «Классические сопрограммы» главы 17 и скриптом *flags_asyncio.py* заключается в том, что во втором случае не видно никаких вызовов `.send()` или выражений `yield`. Наш код находится между библиотекой *asyncio* и асинхронными библиотеками, которыми мы пользуемся, например *HTTPX*. Это показано на рис. 21.1.

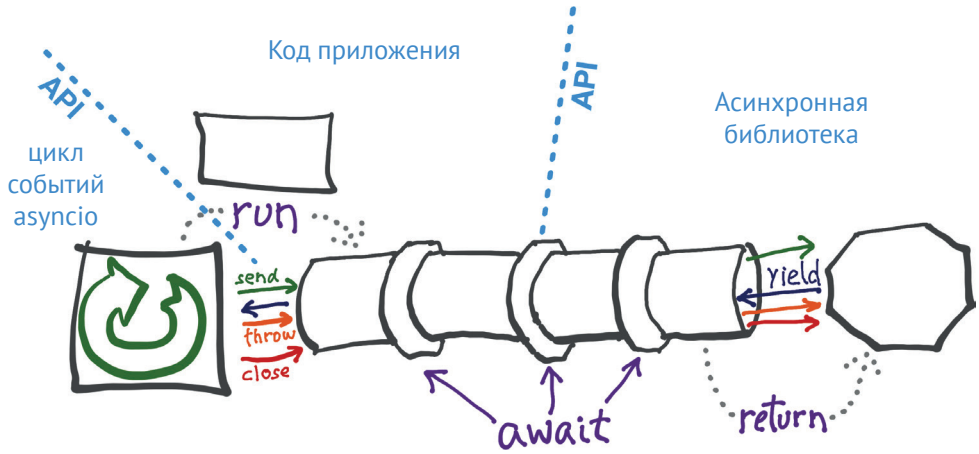


Рис. 21.1. В асинхронной программе пользовательская функция запускает цикл событий, планируя начальную сопрограмму с помощью вызова `asyncio.run`. Каждая пользовательская сопрограмма отдает управление следующей с помощью выражения `await`, формируя канал, по которому взаимодействуют библиотека типа HTTPX и цикл событий

Под капотом цикл событий `asyncio` обращается к `.send`, чтобы привести в действие ваши сопрограммы, а ваши сопрограммы с помощью `await` вызывают другие сопрограммы, в т. ч. библиотечные. Как уже было сказано, `await` заимствует большую часть реализации у предложения `yield from`, которое также обращается к `.send` для управления сопрограммами.

Цепочка `await` в конце концов достигает низкоуровневого объекта, допускающего ожидание, который возвращает генератор, к которому цикл событий может обращаться в ответ на такие события, как срабатывание таймера или сетевой ввод-вывод. Низкоуровневые объекты, допускающие ожидание, и генераторы в конце таких цепочек `await` находятся глубоко внутри библиотек, они не являются частью их API и могут быть расширениями, написанными на C.

Используя функции типа `asyncio.gather` и `asyncio.create_task`, мы можем создать несколько конкурентных каналов `await`, что позволяет конкурентно выполнять несколько операций ввода-вывода в одном цикле событий в одном потоке.

Проблема «все или ничего»

Обратите внимание, что в примере 21.3 я не мог повторно использовать функцию `get_flag` из файла *flags.py* (пример 20.2). Я вынужден был переписать ее в виде сопрограммы, чтобы воспользоваться асинхронным API *HTTPX*. Для достижения максимальной производительности при работе с *asyncio* мы должны заменить все функции, осуществляющие ввод-вывод, асинхронными версиями, которые активируются в результате выполнения `await` или `asyncio.create_task`, для того чтобы управление возвращалось циклу событий, пока функция ждет завершения ввода-вывода. Если вы не можете переписать блокирующую функцию как сопрограмму, то ее следует запускать в отдельном потоке или процессе, как мы увидим в разделе «Делегирование задач исполнителям» ниже. Именно поэтому я и выбрал для этой главы эпиграф, включающий со-

вет: «Вам придется переписать весь код, чтобы нигде ничего не блокировалось, иначе вы просто зря потратите время».

По той же причине я не мог повторно использовать функцию `download_one` из файла *flags_threadpool.py* (пример 20.3). Код в примере 21.3 активирует `get_flag` с помощью `await`, поэтому `download_one` тоже должна быть сопрограммой. Для каждого запроса в функции `supervisor` создается объект сопрограммы `download_one`, и все они управляются сопрограммой `asyncio.gather`.

Теперь изучим предложение `async with`, встречающееся в функциях `supervisor` (пример 21.2) и `get_flag` (пример 21.3).

Асинхронные контекстные менеджеры

В разделе «Контекстные менеджеры и блоки with» главы 18 мы видели, как можно использовать объект, чтобы выполнить некоторый код до и после блока `with`, если его класс предоставляет методы `__enter__` и `__exit__`.

Теперь рассмотрим пример 21.4, взятый из документации по *asyncpg* (*https://magicstack.github.io/asyncpg/current/*) – совместимого с asyncio драйвера PostgreSQL (*https://magicstack.github.io/asyncpg/current/api/index.html#transactions*).

Пример 21.4. Пример из документации по драйверу PostgreSQL asyncpg

```
tr = connection.transaction()
await tr.start()
try:
    await connection.execute("INSERT INTO mytable VALUES (1, 2, 3)")
except:
    await tr.rollback()
    raise
else:
    await tr.commit()
```

Транзакция базы данных естественно ложится на протокол контекстного менеджера: транзакцию нужно начать, изменить данные в `connection.execute`, а затем зафиксировать или откатить в зависимости от того, как прошли изменения.

В асинхронном драйвере типа *asyncpg* процедуры подготовки и завершения должны быть сопрограммами, чтобы остальные операции могли выполняться конкурентно. Однако реализация классического предложения `with` не поддерживает выполнение методов `__enter__` или `__exit__` сопрограммами.

Поэтому в документе PEP 492 «Coroutines with async and await syntax» (*https://peps.python.org/pep-0492/*) было введено предложение `async with`, работающее с асинхронными контекстными менеджерами: объектами, реализующими методы `__aenter__` и `__aexit__` как сопрограммы.

С помощью `async with` пример 21.4 можно переписать в следующем виде, тоже заимствованном из документации по *asyncpg*:

```
async with connection.transaction():
    await connection.execute("INSERT INTO mytable VALUES (1, 2, 3)")
```

В классе `asyncpg.Transaction` (*https://github.com/MagicStack/asyncpg/blob/4d39a05268ce4cc01b00458223a767542da048b8/asyncpg/transaction.py#L57*) метод-сопрограм-

ма `__aenter__` выполняет `await self.start()`, а метод-сопрограмма `__aexit__` ожидает завершения закрытых методов-сопрограмм `__rollback` или `__commit` в зависимости от того, было исключение или нет. Использование сопрограмм для реализации класса `Transaction` как асинхронного контекстного менеджера позволяет *asyncpg* обрабатывать много транзакций конкурентно.

Калеб Хэттинг об asyncpg

У *asyncpg* есть еще одно важное достоинство – он позволяет обойти отсутствие в PostgreSQL поддержки высокой конкурентности (в этой СУБД используется один серверный процесс на каждое подключение), поскольку реализует пул подключений для внутреннего подключения к самой Postgres.

Это означает, что не нужны дополнительные инструменты типа *pgbouncer*, о чем явно написано в документации по *asyncpg* (*https://magicstack.github.io/asyncpg/ current/usage.html#connection-pools*)[1].

Но вернемся к скрипту *flags_asyncio.py*. Класс `AsyncClient` из библиотеки `httpx` является асинхронным контекстным менеджером, поэтому может пользоваться объектами, допускающими ожидание, в своих специальных методах `__aenter__` и `__aexit__`.

В разделе «Асинхронные генераторы как контекстные менеджеры» показано, как использовать библиотеку Python `contextlib` для создания асинхронного контекстного менеджера без написания класса. Это объяснение помещено ниже, потому что ему должен предшествовать раздел «Асинхронные генераторные функции».

Теперь дополним код загрузки флагов с применением *asyncio* индикатором хода выполнения, что даст нам возможность поговорить про другие возможности *asyncio* API.

Улучшение асинхронного загрузчика

В разделе «Загрузка с индикацией хода выполнения и обработкой ошибок» главы 20 все `flags2`-примеры разделяли общий интерфейс командной строки и отображали индикатор хода выполнения во время загрузки. Они также включали обработку ошибок.

Я призываю вас поэкспериментировать с `flags2`-примерами, чтобы интуитивно почувствовать, как ведут себя конкурентные HTTP-клиенты. Флаг `-h` выводит экран справки, как в примере 20.10. Используйте флаги `-a`, `-e` и `-l` для управления количеством загрузок, а флаг `-m` для задания количества конкурентных загрузок. Прогоните тесты, используя серверы `LOCAL`, `REMOTE`, `DELAY` и `ERROR`. Определите оптимальное число конкурентных загрузок, максимизирующее пропускную способность для каждого сервера. Протестируйте параметры тестовых серверов, как описано в примечании «Настройка тестовых серверов» в главе 20.

[1] Этот совет – дословная цитата из комментария технического рецензента Калеба Хэттинга. Спасибо, Калеб!

Так, в примере 21.5 показана попытка получить 100 флагов (`-al 100`) от сервера `ERROR`, используя 100 конкурентных запросов (`-m 100`). 48 ошибок в результате – либо ошибки HTTP 418, либо тайм-ауты – ожидаемое (неправильное) поведение *slow_server.py*.

Пример 21.5. Прогон flags2_asyncio.py

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8002/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
100%|████████████████████████████████████| 100/100 [00:03<00:00, 30.48it/s]
--------------------
 52 flags downloaded.
 48 errors.
Elapsed time: 3.31s
```

Ведите себя ответственно при тестировании конкурентных клиентов

Даже если общее время загрузки для многопоточного и асинхронного HTTP-клиента различается не сильно, асинхронная версия может посылать запросы быстрее, поэтому больше вероятность, что сервер заподозрит DoS-атаку. Чтобы по-настоящему протестировать конкурентные клиенты «на полной тяге», используйте локальные HTTP-серверы, как было описано в примечании «Настройка тестовых серверов» в главе 20.

Теперь посмотрим, как реализован скрипт *flags2_asyncio.py*.

Использование asyncio.as_completed и потока

В примере 21.3 мы передавали несколько сопрограмм функции `asyncio.gather`, которая возвращает список результатов в том порядке, в каком подавались сопрограммы. Это означает, что `asyncio.gather` вернется только после того, как все допускающие ожидание объекты завершат работу. Однако чтобы обновлять индикатор хода выполнения, мы должны получать результаты сразу же.

По счастью, в `asyncio` существует эквивалент генераторной функции `as_completed`, которой мы пользовались в примере пула потоков с индикацией хода выполнения (пример 20.16). В примере 21.6 показано начало скрипта *flags2_asyncio.py*, где определены функции `get_flag` и `download_one`. А в примере 21.7 показана оставшаяся часть исходного кода – функции `supervisor` и `download_many`. Этот скрипт длиннее *flags_asyncio.py* из-за обработки ошибок.

Пример 21.6. flags2_asyncio.py: начало скрипта; продолжение см. в примере 21.7

```
import asyncio
from collections import Counter
from http import HTTPStatus
from pathlib import Path

import httpx
import tqdm # type: ignore
```

```
from flags2_common import main, DownloadStatus, save_flag

# по умолчанию задана низкая степень конкурентности, чтобы избежать ошибок
# удаленного сайта, например 503 - Service Temporarily Unavailable
DEFAULT_CONCUR_REQ = 5
MAX_CONCUR_REQ = 1000

async def get_flag(client: httpx.AsyncClient, ❶
                   base_url: str,
                   cc: str) -> bytes:
    url = f'{base_url}/{cc}/{cc}.gif'.lower()
    resp = await client.get(url, timeout=3.1, follow_redirects=True) ❷
    resp.raise_for_status()
    return resp.content

async def download_one(client: httpx.AsyncClient,
                       cc: str,
                       base_url: str,
                       semaphore: asyncio.Semaphore,
                       verbose: bool) -> DownloadStatus:
    try:
        async with semaphore: ❸
            image = await get_flag(client, base_url, cc)
    except httpx.HTTPStatusError as exc: ❹
        res = exc.response
        if res.status_code == HTTPStatus.NOT_FOUND:
            status = DownloadStatus.NOT_FOUND
            msg = f'not found: {res.url}'
        else:
            raise
   else:
        await asyncio.to_thread(save_flag, image, f'{cc}.gif') ❺
        status = DownloadStatus.OK
        msg = 'OK'
    if verbose and msg:
        print(cc, msg)
    return status
```

❶ Функция `get_flag` очень похожа на последовательную версию из примера 20.14. Первое отличие: она требует параметра `client`.

❷ Второе и третье отличия: `.get` – метод `AsyncClient` и является сопрограммой, поэтому для его выполнения нужно ключевое слово `await`.

❸ Использовать `semaphore` как асинхронный контекстный менеджер, чтобы не блокировать программу целиком; только эта сопрограмма приостанавливается, когда счетчик семафора обращается в нуль. Подробнее см. во врезке «Семафоры в Python» ниже.

❹ Логика обработки ошибок такая же, как в функции `download_one` из примера 20.14.

❺ Сохранение изображения – операция ввода-вывода. Чтобы избежать блокирования цикла событий, функция `save_flag` выполняется в отдельном потоке.

Весь сетевой ввод-вывод в *asyncio* производится с помощью сопрограмм, но к файловому вводу-выводу это не относится. Однако файловый ввод-вывод – тоже блокирующая операция в том смысле, что чтение и запись файлов

занимают в тысячи раз больше времени (*https://gist.github.com/jboner/2841832*), чем чтение-запись в память. А если используется сетевая система хранения (*https://en.wikipedia.org/wiki/Network-attached_storage*), то за кулисами может еще и выполняться сетевой ввод-вывод.

Начиная с Python 3.9 сопрограмма `asyncio.to_thread` упрощает делегирование файлового ввода-вывода пулу потоков, предоставляемому библиотекой *asyncio*. На случай, если нужно поддерживать Python 3.7 или 3.8, в разделе «Делегирование задач исполнителям» показано, как это сделать, добавив всего две строки. Но сначала закончим изучение кода HTTP-клиента.

Регулирование темпа запросов с помощью семафора

Сетевые клиенты типа рассматриваемого здесь следует *дросселировать* (т. е. ограничивать), чтобы избежать затопления сервера слишком большим количеством конкурентных запросов.

Семафор (*https://en.wikipedia.org/wiki/Semaphore_(programming)*) – это примитив синхронизации, более гибкий, чем блокировка. Семафор могут удерживать несколько сопрограмм, причем максимальное их число настраивается. Поэтому это идеальный механизм для ограничения количества активных конкурентных сопрограмм. Дополнительные сведения см. во врезке «Семафоры в Python».

В скрипте *flags2_threadpool.py* (пример 20.16) эффект дросселирования достигался путем создания в функции `download_many` объекта `ThreadPoolExecutor`, в котором обязательный аргумент `max_workers` задавался равным `concur_req`. В скрипте *flags2_asyncio.py* функция `supervisor` (см. пример 21.7) создает экземпляр `asyncio.Semaphore` и передает его в аргументе `semaphore` функции `download_one` (пример 21.6).

Семафоры в Python

Эдсгер Дейкстра изобрел семафор в начале 1960-х годов. Идея простая, но настолько гибкая, что большинство других объектов синхронизации, например блокировки и барьеры, можно построить на основе семафоров. В стандартной библиотеке Python есть три класса `Semaphore`: по одному в модулях `threading`, `multiprocessing` и `asyncio`. Здесь мы рассмотрим последний.

В классе `asyncio.Semaphore` имеется внутренний счетчик, который уменьшается на 1 всякий раз, как выполняется `await` для метода-сопрограммы `.acquire()`, и увеличивается на 1 при вызове метода `.release()`, который не является сопрограммой, потому что никогда не блокирует выполнение. Начальное значение счетчика задается при создании объекта `Semaphore`:

```
semaphore = asyncio.Semaphore(concur_req)
```

Ожидание `.acquire()` не приводит к задержке, когда счетчик больше 0, но если счетчик равен 0, то `.acquire()` приостанавливает ожидающую сопрограмму до тех пор, пока какая-нибудь другая сопрограмма не вызовет `.release()` для того же семафора, увеличив тем самым счетчик. Вместо того чтобы обращаться к этим методам напрямую, безопаснее использовать `semaphore` как асинхронный контекстный менеджер, как я поступил в функции `download_one` из примера 21.6:

```
async with semaphore:
    image = await get_flag(client, base_url, cc)
```

Метод-сопрограмма `Semaphore.__aenter__` ждет завершения `.acquire()`, а метод-сопрограмма `__aexit__` вызывает `.release()`. Этот код гарантирует, что в любой момент времени будет активно не более `concur_req` экземпляров сопрограммы `get_flags`.

У каждого из классов `Semaphore` в стандартной библиотеке имеется подкласс `BoundedSemaphore`, налагающий дополнительное ограничение: внутренний счетчик не может стать больше начального значения, если операций `.release()` окажется больше, чем `.acquire()`[1].

А теперь взглянем на оставшуюся часть скрипта.

Пример 21.7. flags2_asyncio.py: продолжение скрипта из примера 21.6

```
async def supervisor(cc_list: list[str],
                     base_url: str,
                     verbose: bool,
                     concur_req: int) -> Counter[DownloadStatus]: ❶
    counter: Counter[DownloadStatus] = Counter()
    semaphore = asyncio.Semaphore(concur_req) ❷
    async with httpx.AsyncClient() as client:
        to_do = [download_one(client, cc, base_url, semaphore, verbose)
                 for cc in sorted(cc_list)] ❸
        to_do_iter = asyncio.as_completed(to_do) ❹
        if not verbose:
            to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list)) ❺
        error: httpx.HTTPError | None = None ❻
        for coro in to_do_iter: ❼
            try:
                status = await coro ❽
            except httpx.HTTPStatusError as exc:
                error_msg = 'HTTP error {resp.status_code} - {resp.reason_phrase}'
                error_msg = error_msg.format(resp=exc.response)
                error = exc ❾
            except httpx.RequestError as exc:
                error_msg = f'{exc} {type(exc)}'.strip()
                error = exc ❿
            except KeyboardInterrupt:
                break

            if error:
                status = DownloadStatus.ERROR ⓫
                if verbose:
                    url = str(error.request.url) ⓬
                    cc = Path(url).stem.upper() ⓭
                    print(f'{cc} error: {error_msg}')
            counter[status] += 1
    return counter

def download_many(cc_list: list[str],
                  base_url: str,
                  verbose: bool,
                  concur_req: int) -> Counter[DownloadStatus]:
```

[1] Спасибо Гуто Майя, который обратил внимание, что в первом варианте этой главы понятие семафора не объяснялось.

```
    coro = supervisor(cc_list, base_url, verbose, concur_req)
    counts = asyncio.run(coro) ⑭

    return counts
if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

❶ `supervisor` принимает те же аргументы, что функция `download_many`, но ее нельзя вызывать из `main` напрямую, потому что это сопрограмма, а не обычная функция.

❷ Создать семафор `asyncio.Semaphore`, которым смогут одновременно пользоваться не более `concur_req` сопрограмм. Значение `concur_req` вычисляется функцией `main` из модуля *flags2_common.py* на основе параметров командной строки и констант в каждом примере.

❸ Создать список объектов сопрограмм, по одному на каждый вызов сопрограммы `download_one`.

❹ Получить итератор, который будет возвращать объекты сопрограмм по мере их завершения. Я не стал помещать это обращение к `as_completed` в цикл `for` ниже, потому что, возможно, понадобится обернуть его итератором `tqdm` для индикатора хода выполнения – в зависимости от режима, заданного пользователем в командной строке.

❺ Обернуть итератор `as_completed` генераторной функцией `tqdm`, чтобы показать индикатор хода выполнения.

❻ Объявить переменную `error` и инициализировать ее значением `None`; в этой переменной мы будем хранить исключение за пределами предложения `try/except`, если таковое возникнет.

❼ Обойти завершившиеся объекты сопрограмм; этот цикл похож на тот, что использовался в функции `download_many` из примера 20.16.

❽ Ждать завершения сопрограммы для получения результата. Это предложение не приводит к блокированию, потому что `as_completed` порождает только уже завершившиеся сопрограммы.

❾ Это присваивание необходимо, поскольку область видимости переменной `exc` ограничена этой ветвью `except`, а мне нужно сохранить ее значение для будущего использования.

❿ То же, что и выше.

⓫ Если была ошибка, установить переменную `status`.

⓬ В режиме подробной диагностики извлечь URL-адрес из возникшего исключения…

⓭ …и имя файла, чтобы показать код страны.

⓮ `download_many` создает объект сопрограммы `supervisor` и передает его циклу событий, вызвав `asyncio.run`, а затем получает счетчик, который `supervisor` возвращает по завершении цикла событий.

В примере 21.7 мы не могли использовать отображение будущих объектов на коды стран, которое встречали в примере 20.16, потому что допускающие ожидание объекты, возвращенные `asyncio.as_completed`, – это те же объекты, которые мы передали при вызове `as_completed`. Внутренние алгоритмы *asyncio* могут

заменить предоставленные нами объекты другими, но результаты они будут возвращать точно такие же[1].

Поскольку я не мог использовать объекты, допускающие ожидание, для извлечения кода страны из словаря в случае ошибки, я вынужден был извлекать этот код из исключения. Для этого я сохранил исключение в переменной `error`, к которой можно получить доступ извне предложения `try/except`. Python не является языком с блочной областью видимости: такие предложения, как циклы и `try/except`, не создают локальной области видимости в блоках, которыми управляют. Но если в ветви `except` исключение связывается с переменной, как случае переменных `exc` в этом примере, то связь существует только внутри блока, ограниченного этой конкретной ветвью `except`.

На этом мы завершаем обсуждение примера *asyncio*, функционально эквивалентного ранее рассмотренному скрипту *flags2_threadpool.py*.

В следующем примере демонстрируется простой паттерн выполнения одной асинхронной задачи вслед за другой с помощью сопрограмм. Это заслуживает внимания, потому что всякий имеющий опыт работы с JavaScript знает, что выполнение одной асинхронной функции после другой – основная причина паттерна вложенного кода, известного под названием «пирамида судьбы». Ключевое слово `await` снимает это проклятие. Поэтому `await` и вошло в состав Python и JavaScript.

Отправка нескольких запросов при каждой загрузке

Предположим, что мы хотим сохранить вместе с флагом каждой страны ее название и код, а не только код. Тогда нужно отправить два HTTP-запроса на каждый флаг: один для получения самого изображения флага, а другой для получения файла *metadata.json*, находящегося в том же каталоге, что изображение, – именно там хранится название страны.

В многопоточном скрипте координировать несколько запросов в одной задаче просто: нужно лишь отправлять запросы один за другим, дважды блокируя поток, и сохранять оба элемента данных (код и название страны) в локальных переменных, которые понадобятся при сохранении файлов. Если мы захотим сделать то же самое в асинхронном скрипте с помощью обратных вызовов, то понадобятся вложенные функции, так чтобы код и название страны были доступны в их замыканиях до момента сохранения файла, – ведь каждый обратный вызов работает в своей локальной области видимости. Ключевое слово `await` освобождает от этой повинности, поскольку дает возможность запускать асинхронные запросы один за другим в общей локальной области видимости активирующей сопрограммы.

[1] Подробное обсуждение этого момента можно найти в ветке, которую я создал в группе python-tulip. Она называется «Which other futures may come out of asyncio.as_completed?» (*https://asgi.readthedocs.io/en/latest/implementations.html*). Ответ Гвидо проливает свет на реализацию `as_completed`, а также на тесную связь между будущими объектами и сопрограммами в asyncio.

Если при программировании асинхронного приложения вы злоупотребляете обратными вызовами, то, скорее всего, следуете старым паттернам, которые в современном Python не имеют смысла. Это оправдано, если вы пишете библиотеку, которая имеет интерфейсы к унаследованному или низкоуровневому коду, не поддерживающему сопрограммы. Как бы то ни было, в ответе на вопрос «What is the use case for future.add_done_callback()?» (*https://stackoverflow.com/questions/53701841 / what-is-the-use-case-for-future-add-done-callback/53710563*) на сайте StackOverflow объясняется, почему обратные вызовы необходимы в низкоуровневом коде, но не очень полезны в коде современного приложения на Python.

В третьем варианте скрипта загрузки флагов с помощью `asyncio` внесено несколько изменений:

`get_country`

Эта новая сопрограмма скачивает файл *metadata.json*, соответствующий коду страны, и читает из него название страны.

`download_one`

Эта сопрограмма использует `await`, чтобы делегировать работу `get_flag` и новой сопрограмме `get_country` и, пользуясь результатом последней, построить имя сохраняемого файла.

Начнем с кода `get_country` (пример 21.8). Обратите внимание, что он очень похож на код `get_flag` из примера 21.6.

Пример 21.8. flags3_asyncio.py: сопрограмма `get_country`

```
async def get_country(client: httpx.AsyncClient,
                      base_url: str,
                      cc: str) -> str: ❶
    url = f'{base_url}/{cc}/metadata.json'.lower()
    resp = await client.get(url, timeout=3.1, follow_redirects=True)
    resp.raise_for_status()
    metadata = resp.json() ❷
    return metadata['country'] ❸
```

❶ Эта сопрограмма возвращает строку, содержащую название страны, – если все пройдет хорошо.

❷ В `metadata` будет находиться словарь Python, построенный по содержимому ответа в формате JSON.

❸ Вернуть название страны.

Теперь рассмотрим модифицированную сопрограмму `download_one` в примере 21.9; по сравнению с сопрограммой из примера 21.6 в ней изменилось всего несколько строк.

Пример 21.9. flags3_asyncio.py: сопрограмма `download_one`

```
async def download_one(client: httpx.AsyncClient,
                       cc: str,
                       base_url: str,
                       semaphore: asyncio.Semaphore,
                       verbose: bool) -> DownloadStatus:
    try:
        async with semaphore: ❶
            image = await get_flag(client, base_url, cc)
        async with semaphore: ❷
            country = await get_country(client, base_url, cc)
    except httpx.HTTPStatusError as exc:
        res = exc.response
        if res.status_code == HTTPStatus.NOT_FOUND:
            status = DownloadStatus.NOT_FOUND
            msg = f'not found: {res.url}'
        else:
            raise
    else:
        filename = country.replace(' ', '_') ❸
        await asyncio.to_thread(save_flag, image, f'{filename}.gif')
        status = DownloadStatus.OK
        msg = 'OK'
    if verbose and msg:
        print(cc, msg)
    return status
```

❶ Удерживать семафор, чтобы дождаться результата `get_flag` ...

❷ ... и еще раз для ожидания `get_country`.

❸ Использовать название страны для создания имени файла. Работая в основном в режиме командной строки, я не люблю пробелов в именах файлов.

Гораздо лучше обратных вызовов!

Я поместил вызовы `get_flag` и `get_country` в разные блоки `with`, контролируемые семафором, потому что рекомендуется удерживать семафоры и блокировки в течение как можно более короткого времени.

Я мог бы запланировать параллельное выполнение `get_flag` и `get_country`, воспользовавшись `asyncio.gather`, но если `get_flag` возбудит исключение, то нечего будет сохранять, поэтому вызывать `get_country` бессмысленно. Однако бывают ситуации, когда имеет смысл с помощью `asyncio.gather` обратиться к нескольким API одновременно, а не ждать получения одного ответа, прежде чем отправить следующий запрос.

В скрипте *flags3_asyncio.py* ключевое слово `await` встречается шесть раз, а `async with` – три раза. Надеюсь, вы уяснили суть асинхронного программирования в Python. Но нужно понимать, когда необходимо использовать `await`, а когда без этого можно обойтись. В принципе, ответ прост: с помощью `await` нужно ждать завершения сопрограмм и других объектов, допускающих ожидание, например экземпляров класса `asyncio.Task`. Но некоторые API запутаны, в них сопрограммы и обычные функции сочетаются, на первый взгляд, произвольными способами. Так, например, устроен класс `StreamWriter`, которым мы воспользуемся в примере 21.14.

Пример 21.9 – последний из серии скриптов *flags*. Теперь обсудим использование процессов и потоков в асинхронном программировании.

Делегирование задач исполнителям

Важное преимущество Node.js перед Python в плане асинхронного программирования – стандартная библиотека, которая предлагает асинхронные API для всех видов ввода-вывода, а не только для сетевого. В Python беспечность может серьезно подорвать производительность асинхронных приложений, потому что чтение и запись в систему хранения в главном потоке блокируют цикл событий.

В сопрограмме `download_one` из примера 21.6 я написал такую строку для сохранения загруженного изображения на диск:

```
await asyncio.to_thread(save_flag, image, f'{cc}.gif')
```

Как уже было сказано, сопрограмма `asyncio.to_thread` была добавлена в Python 3.9. Если необходимо поддержать версии 3.7 или 3.8, то эту строку можно заменить строками, показанными в примере 21.10.

Пример 21.10. Строки, заменяющие `await asyncio.to_thread`

```
loop = asyncio.get_running_loop()          ❶
loop.run_in_executor(None, save_flag,      ❷
                     image, f'{cc}.gif') ❸
```

❶ Получить ссылку на цикл событий.

❷ Первый аргумент – исполнитель; если передать `None`, то будет выбран экземпляр `ThreadPoolExecutor` по умолчанию, который всегда доступен в цикле событий `asyncio`.

❸ Можно передать подлежащей выполнению функции позиционные аргументы, но если требуется передать именованные, то придется прибегнуть к функции `functool.partial`, как описано в документации по `run_in_executor` (*https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.loop.run_in_ executor*).

Более современная функция `asyncio.to_thread` проще и гибче, поскольку принимает именованные аргументы тоже.

В реализации самого пакета `asyncio` сопрограмма `run_in_executor` используется в нескольких местах. Например, сопрограмма `loop.getaddrinfo(…)`, которую мы видели в примере 21.1, реализована путем вызова функции `getaddrinfo` из модуля `socket`, но это блокирующая функция, которая может работать несколько секунд, поскольку зависит от DNS-сервера.

В асинхронных API часто встречается следующий паттерн: обернуть блокирующий вызов, являющийся деталью реализации, сопрограммой, в которой используется `run_in_executor`. Тем самым мы предоставляем единообразный интерфейс сопрограмм, которым можно управлять с помощью `await`, и скрываем потоки, заведенные по чисто прагматическим причинам. API асинхронного драйвера Motor (*https://motor.readthedocs.io/en/stable/*) для MongoDB, совместимый с `async/await`, на самом деле является фасадом вокруг многопоточного ядра, которое общается с сервером базы данных. А. Джесси Жирью Дэвис, ве-

дущий разработчик Motor, объяснил ход своих рассуждений в статье «Response to 'Asynchronous Python and Databases'» (*https://emptysqua.re/blog/response-to-asynchronous-python-and-databases/*). Спойлер: Дэвис обнаружил, что пул потоков в конкретном случае драйвера базы данных работает быстрее – вопреки мифу, будто асинхронные подходы всегда быстрее потоков при выполнении сетевого ввода-вывода.

Главная причина явной передачи `Executor` функции `loop.run_in_executor` – использовать `ProcessPoolExecutor`, если исполняемая функция счетная; тогда она сможет работать в другом процессе Python, избежав состязания за GIL. Из-за высоких накладных расходов на инициализацию лучше запустить `ProcessPoolExecutor` в `supervisor` и передавать его сопрограммам, которым он нужен.

Калеб Хэттинг, автор книги «Using Asyncio in Python» (O'Reilly) (*https://www.oreilly.com/library/view/using-asyncio-in/9781492075325/*) и один из технических рецензентов этой книги, предложил включить следующее предупреждение, касающееся исполнителей и *asyncio*.

Предупреждение Калеба о run_in_executors

Использование `run_in_executor` может приводить к трудным для отладки проблемам, потому что отмена не всегда работает, как ожидается. Сопрограммы, в которых используются исполнители, только делают вид, что отменились: для стоящего за ними потока (если это `ThreadPoolExecutor`) нет никакого механизма отмены. Например, долго работающий поток, созданный внутри `run_in_executor`, может помешать вашей асинхронной программе чисто завершиться: `asyncio.run` будет ждать, пока исполнитель завершится, перед тем как вернуться, и это будет продолжаться вечно, если задания, выполняемые исполнителем, не завершаются по собственной инициативе. Я по-стариковски хотел бы, чтобы эта функция называлась `run_in_executor_uncancellable`.

Перейдем теперь от клиентских скриптов к написанию серверов с применением `asyncio`.

Написание асинхронных серверов

Классический модельный пример TCP-сервера – эхо-сервер (*https://docs.python.org/3/library/asyncio-stream.html#tcp-echo-server-using-streams*). Мы создадим игрушки поинтереснее: серверные утилиты поиска символов Unicode, сначала на базе HTTP и *FastAPI*, а затем ограничившись чистым TCP с `asyncio`.

Эти серверы позволят пользователям запрашивать символы Unicode, содержащие указанные слова в стандартном имени из модуля `unicodedata`, который мы обсуждали в разделе «База данных Unicode Database» главы 4. На рис. 21.2 показан сеанс работы со скриптом *web_mojifinder.py*, первым нашим сервером.

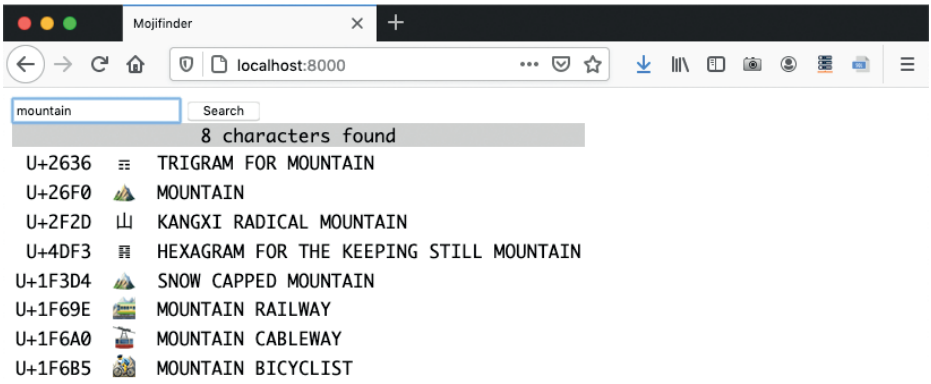


Рис. 21.2. Окно браузера, в котором отображаются результаты поиска по слову «mountain», полученные от службы web_mojifinder.py

Логика поиска символов Unicode в этих примерах инкапсулирована в классе `InvertedIndex`, находящемся в модуле *charindex.py* в репозитории кода на сопроводительном сайте (*https://github.com/fluentpython/example-code-2e*). Никакой конкурентности в этом небольшом модуле нет, поэтому я лишь коротко опишу его во врезке ниже. Можете перейти сразу к реализации HTTP-сервера в разделе «Веб-служба FastAPI Web Service».

Познакомьтесь с инвертированным индексом

Инвертированный индекс обычно отображает слова на документы, в которых они встречаются. В следующих ниже примерах *mojifinder* каждый «документ» – это один символ Unicode. Класс `charindex.InvertedIndex` индексирует все слова, встречающиеся в именах символов в базе данных Unicode, и создает инвертированный индекс, хранящийся в словаре `defaultdict`. Например, чтобы проиндексировать символ U+0037 – DIGIT SEVEN, – инициализатор `InvertedIndex` добавляет символ `'7'` в записи с ключами `'DIGIT'` и `'SEVEN'`. После индексирования данных Unicode 13.0.0, включенных в состав дистрибутива Python 3.9.1, слово `'DIGIT'` отображается на 868 символов, а слово `'SEVEN'` – на 143 символа, включая U+1F556 (CLOCK FACE SEVEN OCLOCK) и U+2790 (DINGBAT NEGATIVE CIRCLED SANS-SERIF DIGIT SEVEN) (он встречается во многих листингах в этой книге).

На рис. 21.3 показаны результаты поиска по словам `'CAT'` и `'FACE'`[1].

[1] Вопросительный знак в квадратике на этом снимке экрана – не дефект печатной или электронной книги. Это символ U+101EC – PHAISTOS DISC SIGN CAT, которого нет в моем терминальном шрифте. Фестский диск – древний артефакт, вырезанный в пиктограммах, найденных на острове Крит.

```
>>> from charindex import InvertedIndex
>>> idx.entries['CAT']
{'😸', '😼', '챨', '🈓', '😺', '😻', '😹', '😽', '😺', '😺', '🐱', '@ ', '😻', '😻'}
>>> len(idx.entries['FACE'])
171
>>> idx.entries['FACE'] & idx.entries['CAT']
{'😸', '😼', '😺', '😹', '😽', '😺', '😺', '😻', '😻', '😻'}
>>> idx.search('cat face')
{'😸', '😼', '😺', '😹', '😽', '😺', '😺', '😻', '😻', '😻'}
>>>
```

Рис. 21.3. Изучение атрибута `entries` и метода `search` класса `InvertedIndex` на консоли Python

Метод `InvertedIndex.search` разбивает запрос на слова и возвращает пересечение найденных для каждого слова множеств символов. Поэтому запрос «face» находит 171 результат, запрос «cat» – 14 результатов, но запрос «cat face» – только 10 результатов.

Красивая идея инвертированного индекса лежит в основе информационного поиска – теории, стоящей за поисковыми системами. См. статью «Inverted Index» (*https://en.wikipedia.org/wiki/Inverted_index*) в англоязычной Википедии.

Веб-служба FastAPI

Следующий пример – *web_mojifinder.py* – я написал с использованием *FastAPI* (*https://fastapi.tiangolo.com/*): одного из написанных на Python веб-каркасов ASGI, упомянутых во врезке «ASGI – шлюзовой интерфейс асинхронных серверов» в главе 19. На рис. 21.2 показан внешний вид интерфейса. Это очень простое одностраничное приложение (Single Page Application – SPA): после начальной загрузки HTML пользовательский интерфейс обновляется клиентским JavaScript-скриптом, взаимодействующим с сервером.

FastAPI предназначена для реализации серверных частей SPA и мобильных приложений, которые состоят по большей части из оконечных точек API, возвращающих ответ в формате JSON, а не в виде сгенерированного сервером HTML-кода. В *FastAPI* вовсю используются декораторы, аннотации типов и интроспекция кода, чтобы устранить трафаретный код из серверных API. Кроме того, автоматически публикуется схема OpenAPI (или Swagger – (*https://swagger.io/specification/*)) – интерактивная документация по созданным нами API. На рис. 21.4 показана автоматически сгенерированная страница */docs* для скрипта *web_mojifinder.py*.

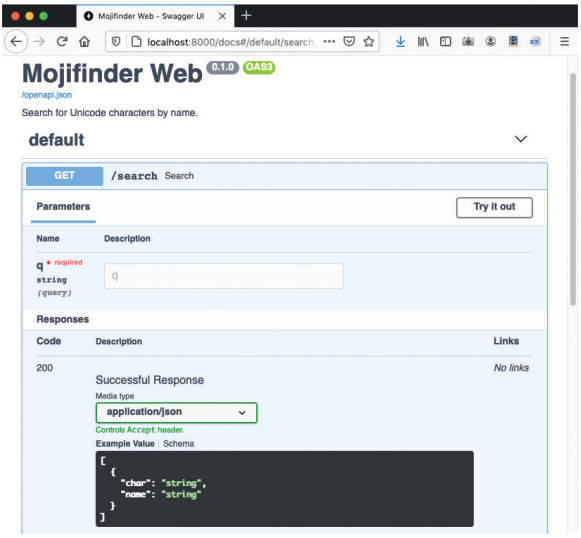


Рис. 21.4. Автоматически сгенерированная схема OpenAPI для оконечной точки `/search`

В примере 21.11 приведен код *web_mojifinder.py*, но это только серверный код. При запросе к корневому URL `/` сервер отправляет файл *form.html*, содержащий 81 строку кода, включая 54 строки на JavaScript для взаимодействия с сервером и заполнения таблицы. Если вам интересно познакомиться с простым JavaScript-кодом, не зависящим ни от какого каркаса, загляните в файл *21-async/mojifinder/static/form.html* в репозитории кода по адресу *https://github.com/fluentpython/example-code-2e*.

Чтобы выполнить скрипт *web_mojifinder.py*, необходимо установить два пакета и их зависимости: *FastAPI* и *uvicorn*[1]. Следующая команда запускает код из примера 21.11 с *uvicorn* в режим разработки:

```
$ uvicorn web_mojifinder:app --reload
```

Опишем параметры:

`web_mojifinder:app`

Имя пакета, двоеточие и имя определенного в нем ASGI-приложения; по соглашению, приложение часто называется `app`.

`--reload`

Поручить *uvicorn* отслеживать изменения в исходных файлах приложения и автоматически перезагружать их. Полезно только на этапе разработки.

Теперь рассмотрим исходный код *web_mojifinder.py*.

[1] Вместо *uvicorn* можно взять другой сервер ASGI, например *hypercorn* или *Daphne*. Дополнительные сведения о реализациях см. в официальной документации по ASGI (*https://asgi.readthedocs.io/en/latest/ implementations.html*).

Пример 21.11. web_mojifinder.py: полный исходный код

```
from pathlib import Path
from unicodedata import name

from fastapi import FastAPI
from fastapi.responses import HTMLResponse
from pydantic import BaseModel

from charindex import InvertedIndex

STATIC_PATH = Path(__file__).parent.absolute() / 'static' ❶

app = FastAPI( ❷
    title='Mojifinder Web',
    description='Search for Unicode characters by name.',
)

class CharName(BaseModel): ❸
    char: str
    name: str

def init(app): ❹
    app.state.index = InvertedIndex()
    app.state.form = (STATIC_PATH / 'form.html').read_text()

init(app) ❺

@app.get('/search', response_model=list[CharName]) ❻
async def search(q: str): ❼
    chars = sorted(app.state.index.search(q)) ❽
    return ({'char': c, 'name': name(c)} for c in chars)

@app.get('/', response_class=HTMLResponse, include_in_schema=False)
def form(): ❾
    return app.state.form ❿

# функции main нет
```

❶ Не относится к теме этой главы, но полезно отметить: элегантное использование перегруженного оператора `/` в модуле `pathlib`[1].

❷ В этой строке определяется ASGI-приложение. Достаточно было бы написать просто `app = FastAPI()`. Показанные параметры – это метаданные для автоматического генерирования документации.

❸ *Пидантическая* схема JSON-ответа с полями `char` и `name`[2].

❹ Построить индекс и загрузить статическую HTML-форму, присоединив то и другое к `app.state` для последующего использования.

❺ Выполнить `init` в момент загрузки этого модуля ASGI-сервером.

❻ Маршрут к оконечной точке `/search`; `response_model` использует *пидантическую* модель `CharName` для описания формата ответа.

[1] Спасибо техническому рецензенту Мирославу Седивы, который подсказал, где в примерах кода стоит использовать `pathlib`.

[2] Как было сказано в главе 8, *пидантичность* (*https://pydantic-docs.helpmanual.io/*) требует проверки аннотаций типов во время выполнения с целью контроля данных.

❼ *FastAPI* предполагает, что параметры, встречающиеся в сигнатуре функции или сопрограммы, но не присутствующие в пути маршрута, передаются в строке HTTP-запроса, например `/search?q=cat`. Поскольку для `q` не задано значение по умолчанию, *FastAPI* вернет код состояния 422 (Unprocessable Entity), если `q` отсутствует в строке запроса.

❽ Возврат итерируемого объекта, состоящего из словарей и совместимого со схемой `response_model`, позволяет *FastAPI* построить JSON-ответ в соответствии со схемой `response_model`, заданной в декораторе `@app.get`.

❾ Обычные (не асинхронные) функции тоже можно использовать для генерирования ответов.

❿ В этом модуле нет функции `main`. Он загружается и выполняется ASGI-сервером – в данном случае *uvicorn*.

В примере 21.11 нет прямых обращений к `asyncio`. *FastAPI* построен на базе ASGI-инструментария *Starlette*, который, в свою очередь, использует `asyncio`.

Отметим также, что в теле `search` не используются `await`, `async with` и `async for`, так что это могла бы быть обычная функция. Я определил `search` как сопрограмму, просто чтобы показать, что *FastAPI* знает, как с ней поступать. В реальном приложении большинство оконечных точек обращаются к базам данных или к другим удаленным серверам, так что для *FastAPI* – и других ASGI-каркасов – критически важна поддержка сопрограмм, которые могут воспользоваться асинхронными библиотеками для сетевого ввода-вывода.

Функции `init` и `form`, написанные мной для загрузки и отдачи статической HTML-формы, нужны только, чтобы сделать пример коротким и простым для выполнения. На практике рекомендуется помещать перед ASGI-сервером прокси-сервер с возможностью балансировки нагрузки, который будет обрабатывать все статические файлы, а также по возможности использовать CDN (сеть доставки контента). Одним из таких прокси-серверов является *Traefik* (*https://doc.traefik.io/traefik/*), описываемый как «граничный маршрутизатор», который «получает запросы от имени вашей системы и определяет, какие компоненты отвечают за их обработку». В состав *FastAPI* входят скрипты генерирования проекта (*https://fastapi.tiangolo.com/project-generation/*), подготавливающие ваш код к работе в таком режиме.

Энтузиаст типизации, возможно, обратил внимание, что в `search` и `form` нет аннотаций типа возвращаемого значения. Вместо этого *FastAPI* полагается на именованный аргумент `response_model=` в маршрутных декораторах. На странице «Модель ответа» (*https://fastapi.tiangolo.com/tutorial/response-model/*) в документации по *FastAPI* приводится такое объяснение:

> Модель ответа объявляется в этом параметре, а не в аннотации типа возвращаемого функцией значения, потому что функция пути может возвращать не саму модель ответа, а словарь, объект базы данных или какую-то другую модель, а затем использовать `response_model`, чтобы наложить ограничения и сериализовать поля.

Например, в маршруте `search` я вернул генератор элементов типа `dict`, а не объектов `CharName`, но этого достаточно для *FastAPI* и *пидантической* схемы,

чтобы проверить мои данные и построить JSON-ответ, совместимый с `response_model=list[CharName]`.

Теперь обратимся к скрипту *tcp_mojifinder.py*, который отвечает на запросы, как показано на рис. 21.5.

Асинхронный TCP-сервер

В программе *tcp_mojifinder.py* используется протокол TCP для взаимодействия с клиентом типа Telnet или Netcat, поэтому я смог написать ее, используя только `asyncio` без внешних зависимостей – и не изобретая заново HTTP. На рис. 21.5 показан пример текстового пользовательского интерфейса.

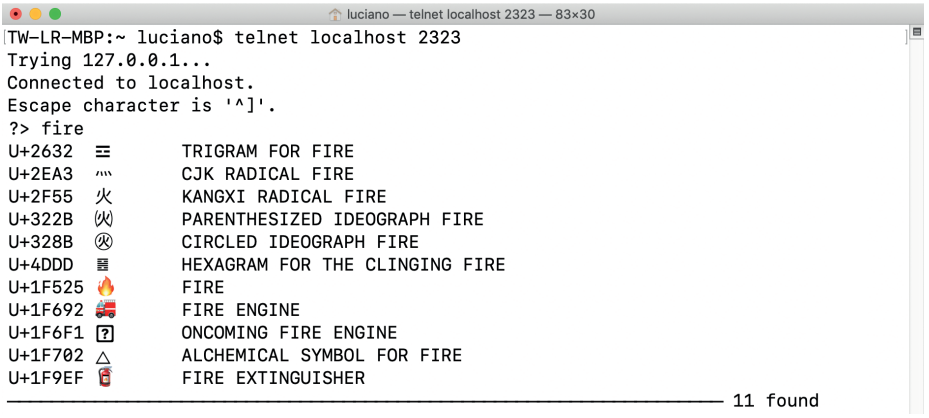


Рис. 21.5. Telnet-сеанс с сервером tcp_mojifinder.py: запрос по слову «fire»

Эта программа в два раза длиннее *web_mojifinder.py*, поэтому я разбил ее описание на три части: примеры 21.12, 21.14 и 21.15. Начало файла *tcp_mojifinder.py*, включая предложения `import`, показано в примере 21.14, но начать я решил с сопрограммы `supervisor` и функции `main`, приводящих программу в действие.

Пример 21.12. tcp_mojifinder.py: простой TCP-сервер; продолжение в примере 21.14

```
async def supervisor(index: InvertedIndex, host: str, port: int) -> None:
    server = await asyncio.start_server( ❶
        functools.partial(finder, index), ❷
        host, port) ❸

    socket_list = cast(tuple[TransportSocket, ...], server.sockets) ❹
    addr = socket_list[0].getsockname()
    print(f'Serving on {addr}. Hit CTRL-C to stop.') ❺
    await server.serve_forever() ❻

def main(host: str = '127.0.0.1', port_arg: str = '2323'):
    port = int(port_arg)
    print('Building index.')
    index = InvertedIndex() ❼
    try:
        asyncio.run(supervisor(index, host, port)) ❽
    except KeyboardInterrupt: ❾
        print('\nServer shut down.')
```

```
if __name__ == '__main__':
    main(*sys.argv[1:])
```

❶ Это `await` быстро получает экземпляр `asyncio.Server`, TCP-сервера. По умолчанию `start_server` создает и запускает сервер, поэтому он готов к приему запросов на подключение.

❷ Первым аргументом `start_server` является `client_connected_cb` – обратный вызов, который выполняется, когда появляется новое клиентское подключение. Это может быть функция или сопрограмма, но в любом случае она должна принимать ровно два аргумента: `asyncio.StreamReader` и `asyncio.StreamWriter`. Однако моей сопрограмме `finder` нужен также индекс, поэтому я воспользовался функцией `functools.partial`, чтобы привязать данный параметр и получить вызываемый объект, принимающий читателя и писателя. Адаптация пользовательских функций к API обратных вызовов – самое частое применение `functools.partial`.

❸ `host` и `port` – второй и третий аргументы `start_server`. Полную сигнатуру см. в документации по *asyncio* (*https://docs.python.org/3/library/asyncio-stream.html#asyncio.start_server*).

❹ Этот `cast` необходим, потому что в *typeshed* находится устаревшая аннотация типа для свойства `sockets` класса `Server` – по состоянию на май 2021 года. См. проблему #5535 в *typeshed* (*https://github.com/python/typeshed/issues/5535*)[1].

❺ Показать адрес и порт первого сокета сервера.

❻ Хотя `start_server` уже запустила сервер как конкурентную задачу, я должен добавить `await` при вызове метода `server_forever`, чтобы моя сопрограмма `supervisor` здесь приостановилась. Иначе `supervisor` вернулась бы немедленно, что привело бы к завершению цикла, начатого вызовом `asyncio.run(supervisor(…))`, и выходу из программы. В документации по `Server.serve_forever` (*https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.Server.serve_forever*) написано: «Этот метод можно вызывать, если сервер уже принимает запросы на подключение».

❼ Построить инвертированный индекс[2].

❽ Запустить цикл событий, в котором исполняется `supervisor`.

❾ Перехватить исключение `KeyboardInterrupt`, чтобы при остановке сервера путем нажатия **Ctrl-C** в терминале, где он исполняется, не показывалась отвлекающая внимание трасса вызовов.

Понять, как устроен поток управления в *tcp_mojifinder.py*, будет проще, изучив распечатку на консоли сервера (см. пример 21.13).

---

[1] Проблема #5535 была закрыта в октябре 2021 года, но с тех пор новых версий Муру не выходило, поэтому ошибка остается.

[2] Технический рецензент Леонардо Рохаэль отметил, что построение индекса можно было бы вынести в другой поток, вызвав `loop.run_with_executor()` в сопрограмме `supervisor`; тогда сервер был бы готов принимать запросы немедленно, пока индекс строится. Это правда, но обращение к индексу – единственное, что этот сервер делает, так что в данном примере мы бы ничего не выиграли.

Пример 21.13. tcp_mojifinder.py: это серверная часть сеанса, изображенного на рис. 21.5

```
$ python3 tcp_mojifinder.py
Building index. ❶
Serving on ('127.0.0.1', 2323). Hit Ctrl-C to stop.❷
From ('127.0.0.1', 58192): 'cat face' ❸
To ('127.0.0.1', 58192): 10 results.
From ('127.0.0.1', 58192): 'fire' ❹
To ('127.0.0.1', 58192): 11 results.
From ('127.0.0.1', 58192): '\x00' ❺
Close ('127.0.0.1', 58192). ❻
^C ❼
Server shut down. ❽
$
```

❶ Напечатано `main`. До появления следующей строки я наблюдал задержку 0,6 с; в это время строился индекс.
❷ Напечатано `supervisor`.
❸ Первая итерация цикла `while` в `finder`. Стек TCP/IP назначил моему Telnet-клиенту порт 58192. Если к серверу подключилось несколько клиентов, то в распечатке будут встречаться разные порты.
❹ Вторая итерация цикла `while` в `finder`.
❺ Я нажал в окне клиентского терминала **Ctrl-C**; выход из цикла `while` в `finder`.
❻ Сопрограмма `finder` печатает это сообщение, затем завершается. Тем временем сервер продолжает работать и готов обслужить другого клиента.
❼ Я нажал **Ctrl-C** в окне серверного терминала; `server.serve_forever` прервана, при этом завершается `supervisor` и цикл событий.
❽ Напечатано `main`.

После того как `main` построила индекс и запустила цикл событий, `supervisor` печатает сообщение `Serving on…` и приостанавливается в строке `await server.serve_forever()`. В этот момент управление попадает в цикл событий и там и остается, время от времени возвращаясь в сопрограмму `finder`, которая вновь уступает управление циклу событий всякий раз, как должна дождаться отправки или получения данных из сети.

Пока цикл событий работает, для каждого клиента, подключившегося к серверу, будет создаваться новый экземпляр сопрограммы `finder`. Таким образом, этот простой сервер может одновременно обслуживать много клиентов. Он продолжает работать, пока не возникнет исключение `KeyboardInterrupt` или процесс не будет снят ОС.

Теперь рассмотрим начало скрипта *tcp_mojifinder.py*, где находится сопрограмма `finder`.

Пример 21.14. tcp_mojifinder.py: продолжение примера 21.12

```
import asyncio
import functools
import sys
from asyncio.trsock import TransportSocket
from typing import cast

from charindex import InvertedIndex, format_results ❶
```

```
CRLF = b'\r\n'
PROMPT = b'?> '

async def finder(index: InvertedIndex, ❷
                 reader: asyncio.StreamReader,
                 writer: asyncio.StreamWriter) -> None:
    client = writer.get_extra_info('peername') ❸
    while True: ❹
        writer.write(PROMPT) # не может быть await! ❺
        await writer.drain() # должно быть await! ❻
        data = await reader.readline() ❼
        if not data: ❽
            break
        try:
            query = data.decode().strip() ❾
        except UnicodeDecodeError: ❿
            query = '\x00'
        print(f' From {client}: {query!r}') ⓫
        if query:
            if ord(query[:1]) < 32: ⓬
                break
            results = await search(query, index, writer) ⓭
            print(f' To {client}: {results} results.') ⓮

    writer.close() ⓯
    await writer.wait_closed() ⓰
    print(f'Close {client}.') ⓱
```

❶ `format_results` полезна для отображения результатов вызова `InvertedIndex.search` в текстовом интерфейсе, например на командной консоли или в сеансе Telnet.

❷ Чтобы передать `finder` сопрограмме `asyncio.start_server`, я обернул ее функцией `functools.partial`, потому что сервер ожидает получить сопрограмму или функцию, принимающую только аргументы `reader` и `writer`.

❸ Получить адрес удаленного клиента, подключившегося к сокету.

❹ В этом цикле происходит диалог, который завершается, когда от клиента будет получен управляющий символ.

❺ Метод `StreamWriter.write` – не сопрограмма, а обычная функция; в этой строке посылается приглашение `?>`.

❻ `StreamWriter.drain` сбрасывает буфер `writer`; это сопрограмма, поэтому ей должно предшествовать слово `await`.

❼ `StreamWriter.readline` – сопрограмма, которая возвращает `bytes`.

❽ Если не было получено ни одного байта, значит, клиент закрыл соединение, поэтому выходим из цикла.

❾ Декодировать `bytes` в `str`, пользуясь кодировкой UTF-8 по умолчанию.

❿ Ошибка `UnicodeDecodeError` может возникнуть, когда пользователь нажал **Ctrl-C** и Telnet-клиент отправил управляющие символы; если такое случилось, для простоты заменить запрос нулевым символом.

⓫ Напечатать запрос на консоли.

⓬ Выйти из цикла, если получен управляющий или нулевой символ.

⓭ Выполнить поиск; код представлен в следующем примере.

⓮ Напечатать ответ на консоли.
⓯ Закрыть `StreamWriter`.
⓰ Дождаться закрытия `StreamWriter`. Это рекомендуется в документации по методу `.close()` (*https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamWriter.close*).
⓱ Напечатать на консоли сообщение о завершении этого сеанса с клиентом.

Последняя часть данного примера – сопрограмма `search` (пример 21.15).

Пример 21.15. tcp_mojifinder.py: сопрограмма `search`

```
async def search(query: str, ❶
                 index: InvertedIndex,
                 writer: asyncio.StreamWriter) -> int:
    chars = index.search(query) ❷
    lines = (line.encode() + CRLF for line ❸
             in format_results(chars))
    writer.writelines(lines) ❹
    await writer.drain() ❺
    status_line = f'{"-" * 66} {len(chars)} found' ❻
    writer.write(status_line.encode() + CRLF)
    await writer.drain()
    return len(chars)
```

❶ `search` должна быть сопрограммой, потому что пишет в `StreamWriter` и должна использовать его метод-сопрограмму `.drain()`.
❷ Опросить инвертированный индекс.
❸ Это генераторное выражение будет отдавать байтовые строки в кодировке UTF-8, содержащие кодовую позицию Unicode, сам символ, его имя и последовательность `CRLF`, например `b'U+0039\t9\tDIGIT NINE\r\n'`.
❹ Отправить `lines`. Как ни странно, `writer.writelines` – не сопрограмма.
❺ Но `writer.drain()` – сопрограмма. Не забудем `await`!
❻ Построить и отправить строку состояния.

Отметим, что весь сетевой ввод-вывод в *tcp_mojifinder.py* производится в байтах (`bytes`); мы должны декодировать байты, поступающие из сети, и закодировать строки перед их отправкой. В Python 3 по умолчанию подразумевается кодировка UTF-8, именно ей я неявно пользуюсь в вызовах `encode` и `decode`.

Заметим, что некоторые методы ввода-вывода являются сопрограммами, так что им должно предшествовать слово `await`, тогда как другие – обычные функции. Например, `StreamWriter.write` – обычная функция, потому что она записывает в буфер. С другой стороны, `StreamWriter.drain` сбрасывает буфер и выполняет сетевой ввод-вывод, поэтому является сопрограммой, как и `StreamReader.readline` – но не `StreamWriter.writelines`! Когда я работал над первым изданием книги, документация по `asyncio` API была улучшена – в ней ясно сказано, что является сопрограммами, а что нет (*https://docs.python.org/3/library/asyncio-stream.html#streamwriter*).

В коде *tcp_mojifinder.py* используется высокоуровневый асинхронный API потоков (*https://docs.python.org/3/library/asyncio-stream.html*), который предоставляет готовый сервер, так что вам остается только реализовать функцию-обработчик в виде простого обратного вызова или сопрограммы. Существует также низкоуровневый API транспорта и протоколов (*https://docs.python.org/3/library/asyncio-protocol.html*), на который оказали влияние абстракции транспорта и протоколов в каркасе *Twisted*. Дополнительную информацию, в т. ч. об эхо-серверах и их клиентах на протоколах TCP и UDP, реализованных с помощью этого низкоуровневого API, см. в документации по `asyncio` (*https://docs.python.org/3/library/asyncio-protocol.html#tcp-echo-server*).

А нашей следующей темой будет конструкция `async for` и объекты, приводящие ее в действие.

Асинхронные итераторы и итерируемые объекты

В разделе «Асинхронные контекстные менеджеры» выше мы видели, как `async with` работает с объектами, реализующими методы `__aenter__` и `__aexit__`, которые возвращают объекты, допускающие ожидание, обычно в форме объектов сопрограмм.

Аналогично `async for` работает с *асинхронными итерируемыми объектами*, т. е. с объектами, реализующими метод `__aiter__`. Однако `__aiter__` должен быть обычным методом, а не сопрограммой, и возвращать *асинхронный итератор*.

Асинхронный итератор предоставляет метод-сопрограмму `__anext__`, который возвращает допускающий ожидание объект, чаще всего объект сопрограммы. Ожидается также, что он реализует метод `__aiter__`, который обычно возвращает `self`. Это отражает важное различие между итерируемыми объектами и итераторами, которое обсуждалось в разделе «Не делайте итерируемый объект итератором для самого себя» главы 17.

В документации по асинхронному драйверу PostgreSQL *aiopg* есть пример, иллюстрирующий использование `async for` для итерирования по строкам курсора базы данных:

```
async def go():
    pool = await aiopg.create_pool(dsn)
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            await cur.execute("SELECT 1")
            ret = []
            async for row in cur:
                ret.append(row)
            assert ret == [(1,)]
```

В этом примере запрос возвращает одну строку, но в реалистичном сценарии ответ на запрос `SELECT` может содержать тысячи строк. Для больших ответов в курсор не загружаются все строки сразу. Поэтому важно, чтобы конструкция `async for row in cur:` не блокировала цикл событий, пока курсор ожидает выборки дополнительных строк. Реализовав курсор как асинхронный итератор, *aiopg* может уступать управление циклу событий при каждом вызове `__anext__` и возобновлять работу, когда от PostgreSQL поступят дополнительные строки.

Асинхронные генераторные функции

Для реализации асинхронного итератора нужно написать класс с методами `__anext__` и `__aiter__`, но есть способ проще: написать функцию, объявленную как `async def` и содержащую в теле `yield`. Тут просматривается параллель с тем, как генераторные функции упрощают классический паттерн Итератор.

Рассмотрим простой пример использования `async for` и реализации асинхронного генератора. В примере 21.1 мы видели скрипт *blogdom.py*, который проверял доменные имена. Теперь предположим, что мы нашли другие применения написанной там сопрограмме `probe` и решили поместить ее в новый модуль, *domainlib.py*, вместе с новым асинхронным генератором `multi_probe`, который принимает список доменных имен и отдает результаты по мере завершения проверки.

Вскоре мы покажем реализацию *domainlib.py*, но сначала посмотрим, как она используется в сочетании с новой асинхронной консолью Python.

Эксперименты с асинхронной консолью Python

Начиная с версии Python 3.8 (*https://docs.python.org/3/whatsnew/3.8.html#asyncio*) интерпретатор можно запускать с флагом `-m asyncio`, чтобы получить «асинхронный цикл REPL»: консоль Python, которая импортирует `asyncio`, предоставляет цикл событий и принимает конструкции `await`, `async for` и `async with` в ответ на приглашение верхнего уровня. Иначе использование этих конструкций вне платформенных сопрограмм считалось бы синтаксической ошибкой[1].

Для экспериментов с *domainlib.py* зайдите в каталог *21-async/domains/asyncio/* в своей локальной копии репозитория кода (*https://github.com/fluentpython/example-code-2e*). Затем выполните команду

```
$ python -m asyncio
```

Будет выведено начальное сообщение вида

```
asyncio REPL 3.9.1 (v3.9.1:1e5d33e9b9, Dec 7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>>
```

Обратите внимание: в заголовке сказано, что можно использовать `await` вместо `asyncio.run()` для управления сопрограммами и другими допускающими ожидание объектами. И еще: я не вводил `import asyncio`. Модуль `asyncio` импортируется автоматически, и эта строка ясно сообщает об этом факте пользователю.

Теперь импортируем модуль *domainlib.py* и поэкспериментируем с двумя его сопрограммами: `probe` и `multi_probe` (пример 21.16).

[1] Отличный инструмент для экспериментов, похожий на консоль Node.js. Спасибо Юрию Селиванову за еще один великолепный вклад в асинхронный Python.

Пример 21.16. Эксперименты с domainlib.py после запуска `python3 -m asyncio`

```
>>> await asyncio.sleep(3, 'Rise and shine!') ❶
'Rise and shine!'
>>> from domainlib import *
>>> await probe('python.org') ❷
Result(domain='python.org', found=True) ❸
>>> names = 'python.org rust-lang.org golang.org no-lang.invalid'.split() ❹
>>> async for result in multi_probe(names): ❺
...     print(*result, sep='\t')
...
golang.org      True ❻
no-lang.invalid False
python.org      True
rust-lang.org   True
>>>
```

❶ Пробуем простое `await`, чтобы посмотреть на асинхронную консоль в действии. Совет: сопрограмма `asyncio.sleep()` принимает факультативный второй аргумент, который возвращается, если ожидать ее завершения с помощью `await`.

❷ Выполнить сопрограмму `probe`.

❸ Версия `probe` из `domainlib` возвращает именованный кортеж `Result`.

❹ Построить список доменных имен. Домен верхнего уровня `.invalid` зарезервирован для тестирования. DNS-запросы к таким доменам всегда возвращают ответ NXDOMAIN от сервера, что означает «домен не существует»[1].

❺ С помощью `async for` обойти асинхронный генератор `multi_probe` и напечатать результаты.

❻ Отметим, что результаты печатаются не в том порядке, в каком доменные имена передавались `multiprobe`, а в порядке получения ответов от DNS-сервера.

Из примера 21.16 видно, что `multi_probe` – асинхронный генератор, потому что он совместим с `async for`. Теперь проделаем еще несколько экспериментов, продолжая начатый пример.

Пример 21.17. Дополнительные эксперименты, продолжение примера 21.16

```
>>> probe('python.org') ❶
<coroutine object probe at 0x10e313740>
>>> multi_probe(names) ❷
<async_generator object multi_probe at 0x10e246b80>
>>> for r in multi_probe(names): ❸
...     print(r)
...
Traceback (most recent call last):
  ...
TypeError: 'async_generator' object is not iterable
```

❶ Вызов платформенной сопрограммы дает объект сопрограммы.

[1] См. RFC 6761 «Special-Use Domain Names» (*https://www.artima.com/weblogs/viewpost.jsp?thread=299551*).

❷ Вызов асинхронного генератора дает объект `async_generator`.
❸ Мы не можем использовать обычный цикл `for` с асинхронными генераторами, потому что они реализуют метод `__aiter__`, а не `__iter__`.

Асинхронные генераторы приводятся в действие конструкцией `async for`, которая может быть предложением блока (как в примере 21.16), а также появляться внутри асинхронных включений, которые мы скоро рассмотрим.

Реализация асинхронного генератора

Теперь рассмотрим код *domainlib.py*, содержащий асинхронный генератор `multi_probe` (пример 21.18).

Пример 21.18. domainlib.py: функции для проверки доменных имен

```
import asyncio
import socket
from collections.abc import Iterable, AsyncIterator
from typing import NamedTuple, Optional

class Result(NamedTuple): ❶
    domain: str
    found: bool

OptionalLoop = Optional[asyncio.AbstractEventLoop] ❷

async def probe(domain: str, loop: OptionalLoop = None) -> Result: ❸
    if loop is None:
        loop = asyncio.get_running_loop()

    try:
        await loop.getaddrinfo(domain, None)
    except socket.gaierror:
        return Result(domain, False)
    return Result(domain, True)

async def multi_probe(domains: Iterable[str]) -> AsyncIterator[Result]: ❹
    loop = asyncio.get_running_loop()
    coros = [probe(domain, loop) for domain in domains] ❺
    for coro in asyncio.as_completed(coros): ❻
        result = await coro ❼
        yield result ❽
```

❶ Благодаря `NamedTuple` результат `probe` проще читать и отлаживать.
❷ Этот псевдоним типа создан для того, чтобы следующая строка не оказалась слишком длинной для размещения на печатной странице.
❸ `probe` теперь получает факультативный аргумент `loop`, чтобы избежать повторного вызова `get_running_loop`, когда эта сопрограмма вызывается из `multi_probe`.
❹ Асинхронная генераторная функция порождает асинхронный объект-генератор, который можно аннотировать типом `AsyncIterator[SomeType]`.
❺ Построить список объектов сопрограммы `probe`, по одному для каждого домена.
❻ Здесь `async for` не нужен, т. к. `asyncio.as_completed` – классический генератор.

❼ Ждать завершения объекта сопрограммы для получения результата.
❽ Отдать `result`. Эта строка превращает `multi_probe` в асинхронный генератор.

Цикл `for` в примере 21.18 можно было бы сделать короче:

```
for coro in asyncio.as_completed(coros):
    yield await coro
```

Python разбирает этот код как `yield (await coro)`, поэтому он работает.

Я подумал, что такая краткая запись в первом же примере асинхронного генератора может оказаться непонятной, поэтому разбил ее на две строки.

Имея *domainlib.py*, мы можем продемонстрировать использование асинхронного генератора `multi_probe` в скрипте *domaincheck.py*, который принимает суффикс доменного имени и ищет домены, образованные короткими ключевыми словами Python.

Ниже приведен пример вывода *domaincheck.py*:

```
$ ./domaincheck.py net
FOUND           NOT FOUND
=====           =========
in.net
del.net
true.net
for.net
is.net
                  none.net
try.net
                  from.net
and.net
or.net
else.net
with.net
if.net
as.net
                  elif.net
                  pass.net
                  not.net
                  def.net
```

Благодаря *domainlib* код *domaincheck.py* оказался линейным, как показано в примере 21.19.

Пример 21.19. domaincheck.py: утилита для проверки доменных имен с использованием модуля domainlib

```
#!/usr/bin/env python3
import asyncio
import sys
from keyword import kwlist

from domainlib import multi_probe

async def main(tld: str) -> None:
```

```
    tld = tld.strip('.')
    names = (kw for kw in kwlist if len(kw) <= 4) ❶
    domains = (f'{name}.{tld}'.lower() for name in names) ❷
    print('FOUND\t\tNOT FOUND') ❸
    print('=====\t\t=========')
    async for domain, found in multi_probe(domains): ❹
        indent = '' if found else '\t\t' ❺
        print(f'{indent}{domain}')

if __name__ == '__main__':
    if len(sys.argv) == 2:
        asyncio.run(main(sys.argv[1])) ❻
    else:
       print('Please provide a TLD.', f'Example: {sys.argv[0]} COM.BR')
```

❶ Построить список ключевых слов длины не более 4.
❷ Построить список доменных имен с заданным суффиксом в качестве домена верхнего уровня (TLD).
❸ Отформатировать заголовок для вывода таблицы.
❹ Асинхронно обойти `multi_probe(domains)`.
❺ Задать отступ `indent` шириной 0 или два табулятора, чтобы поместить результат в нужный столбец.
❻ Выполнить сопрограмму `main` с заданным аргументом в командной строке.

У генераторов есть еще одно применение, не связанное с итерированием: их можно использовать в роли контекстных менеджеров. Это относится и к асинхронным генераторам.

Асинхронные генераторы в качестве контекстных менеджеров

Писать свой контекстный менеджер программисту приходится нечасто, но если такая необходимость возникла, вспомните о декораторе `@asynccontextmanager` (*https://docs.python.org/3/library/contextlib.html#contextlib.asynccontextmanager*), добавленном в модуль `contextlib` в версии Python 3.7. Он очень похож на декоратор `@contextmanager`, который мы изучали в разделе «Использование @contextmanager» главы 18.

Интересный пример совместного использования `@asynccontextmanager` и `loop.run_in_executor` приведен в книге Калеба Хэттинга «Using Asyncio in Python». Пример 21.20 взят из кода Калеба – с одним изменением и добавленными выносками.

Пример 21.20. Совместное использование `@asynccontextmanager` и `loop.run_in_executor`

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def web_page(url): ❶
    loop = asyncio.get_running_loop() ❷
    data = await loop.run_in_executor( ❸
        None, download_webpage, url)
    yield data ❹
    await loop.run_in_executor(None, update_stats, url) ❺
```

```
async with web_page('google.com') as data: ❻
    process(data)
```

❶ Декорированная функция должна быть асинхронным генератором.
❷ Небольшое изменение кода Калеба: используется облегченная `get_running_loop` вместо `get_event_loop`
❸ Предположим, что `download_webpage` – блокирующая функция, в которой используется библиотека *requests*; мы выполняем ее в отдельном потоке, чтобы не блокировать цикл событий.
❹ Все строки перед этим выражением `yield` станут методом-сопрограммой `__aenter__` асинхронного контекстного менеджера, построенного декоратором. Значение `data` будет связано с переменной `data` после слова `as` в предложении `async with` ниже.
❺ Строки после `yield` станут методом-сопрограммой `__aexit__`. Здесь еще один блокирующий вызов делегируется исполнителю, работающему в отдельном потоке.
❻ Использовать `web_page` в сочетании с `async with`.

Это сильно напоминает последовательный декоратор `@contextmanager`. Детали посмотрите в разделе «Использование @contextmanager» главы 18, где показана также обработка ошибок в строке `yield`. Еще один пример `@asynccontextmanager` см. в документации по модулю `contextlib` (*https://docs.python.org/3/library/contextlib.html# contextlib.asynccontextmanager*).

Теперь подведем итог обсуждению асинхронных генераторных функций, сравнив их с платформенными сопрограммами.

Асинхронные генераторы и платформенные сопрограммы

Перечислим сходства и различия платформенных сопрограмм и асинхронных генераторных функций.

- Те и другие объявляются с помощью `async def`.
- В теле асинхронного генератора всегда имеется выражение `yield` – оно и делает его генератором. В платформенной сопрограмме `yield` никогда не встречается.
- Платформенная сопрограмма может возвращать с помощью `return` значение, отличное от `None`. В асинхронном генераторе возможны только предложения `return` без указания возвращаемого значения.
- Платформенные сопрограммы – это допускающие ожидание объекты: они могут активироваться выражениями `await` или передаваться многочисленным функциям в модуле `asyncio`, которые принимают допускающие ожидание аргументы, например `create_task`. Асинхронные генераторы не допускают ожидания. Они являются асинхронными итерируемыми объектами и активируются с помощью `async for` или асинхронных включений.

Пришла пора поговорить об асинхронных включениях.

Асинхронные включения и асинхронные генераторные выражения

В документе PEP 530 «Asynchronous Comprehensions» (*https://peps.python.org/pep-0530/*) описана конструкция `async for` и использование `await` во включениях и генераторных выражениях. Эти предложения были реализованы в версии Python 3.6.

Единственная конструкция, определенная в PEP 530, которая может находиться вне тела `async def`, – асинхронное генераторное выражение.

Определение и использование асинхронного генераторного выражения

Имея асинхронный генератор `multi_probe` из примера 21.18, мы могли бы написать еще один асинхронный генератор, возвращающий только имена найденных доменов. Вот как это делается – снова с применением асинхронной консоли, запускаемой с флагом `-m asyncio`:

```
>>> from domainlib import multi_probe
>>> names = 'python.org rust-lang.org golang.org no-lang.invalid'.split()
>>> gen_found = (name async for name, found in multi_probe(names) if found) ❶
>>> gen_found
<async_generator object <genexpr> at 0x10a8f9700> ❷
>>> async for name in gen_found: ❸
...     print(name)
...
golang.org
python.org
rust-lang.org
```

❶ Наличие `async for` превращает это в асинхронное генераторное выражение. Его можно определить в любом месте Python-модуля.

❷ Асинхронное генераторное выражение строит объект `async_generator` – точно того же вида, который возвращает асинхронная генераторная функция вроде `multi_probe`.

❸ Асинхронный объект-генератор активируется предложением `async for`, которое, в свою очередь, может находиться только внутри тела `async def` или на магической асинхронной консоли, которую я использовал в этом примере.

Подведем итоги: асинхронное генераторное выражение может быть определено в любом месте программы, но потребляться может только внутри платформенной сопрограммы или асинхронной генераторной функции.

Остальные конструкции, введенные в PEP 530, можно определить и использовать только внутри платформенных сопрограмм или асинхронных генераторных функций.

Асинхронные включения

Юрий Селиванов, автор документа PEP 530, обосновывает потребность в асинхронных включениях тремя короткими фрагментами кода, которые воспроизводятся ниже.

Все мы согласимся, что хорошо бы иметь возможность переписать следующий код:

```
result = []
async for i in aiter():
    if i % 2:
        result.append(i)
```

в таком виде:

```
result = [i async for i in aiter() if i % 2]
```

Кроме того, если `fun` – платформенная сопрограмма, то хорошо бы иметь возможность написать:

```
result = [await fun() for fun in funcs]
```

Использование `await` в списковом включении аналогично использованию `asyncio.gather`. Но `gather` дает больше контроля над обработкой исключений благодаря факультативному аргументу `return_exceptions`. Калеб Хэттинг рекомендует всегда задавать `return_exceptions=True` (по умолчанию этот аргумент равен `False`). Дополнительные сведения см. в документации по `asyncio.gather` (*https://docs.python.org/3/library/asyncio-task.html#asyncio.gather*).

Вернемся к магической асинхронной консоли:

```
>>> names = 'python.org rust-lang.org golang.org no-lang.invalid'.split()
>>> names = sorted(names)
>>> coros = [probe(name) for name in names]
>>> await asyncio.gather(*coros)
[Result(domain='golang.org', found=True),
Result(domain='no-lang.invalid', found=False),
Result(domain='python.org', found=True),
Result(domain='rust-lang.org', found=True)]
>>> [await probe(name) for name in names]
[Result(domain='golang.org', found=True),
Result(domain='no-lang.invalid', found=False),
Result(domain='python.org', found=True),
Result(domain='rust-lang.org', found=True)]
>>>
```

Я отсортировал список имен, чтобы продемонстрировать, что в обоих случаях результаты выдаются в том порядке, в каком были поданы на вход.

Документ PEP 530 позволяет использовать `async for` и `await` в списковых включениях, равно как в словарных и множественных. Например, ниже показано использование словарного включения для хранения результатов `multi_probe` на асинхронной консоли:

```
>>> {name: found async for name, found in multi_probe(names)}
{'golang.org': True, 'python.org': True, 'no-lang.invalid': False,
'rust-lang.org': True}
```

Мы можем использовать ключевое слово `await` в выражении перед конструкциями `for` или `async for`, а также после ключевого слова `if`. Ниже приведен пример множественного включения для сбора только уже существующих доменных имен:

```
>>> {name for name in names if (await probe(name)).found}
{'rust-lang.org', 'python.org', 'golang.org'}
```

Мне пришлось поставить дополнительные скобки вокруг выражения `await` из-за более высокого приоритета оператора . (точка), поддерживаемого методом `__getattr__`.

Все включения также могут встречаться только в теле `async def` или на асинхронной консоли.

Теперь поговорим об очень важном свойстве предложений `async`, выражений `async` и создаваемых ими объектов. Эти конструкции часто используются в сочетании с *asyncio*, но на самом деле они не зависят от конкретной библиотеки.

async за пределами asyncio: Curio

Языковые конструкции `async/await` в Python не привязаны к конкретному циклу событий или библиотеке[1]. Благодаря расширяемому API, обеспечиваемому специальными методами, любой достаточно мотивированный программист может написать свою среду асинхронного выполнения и каркас для управления платформенными сопрограммами, асинхронными генераторами и т. д.

Именно это и совершил Дэвид Бизли в своем проекте *Curio* (*https://curio.readthedocs.io/en/latest/index.html*). Ему было интересно, как эти новые языковые средства можно использовать в каркасе, созданном с нуля. Напомним, что библиотека `asyncio` была включена в версию Python 3.4 и использовала `yield from` вместо `await`, так что ее API не допускал асинхронных контекстных менеджеров, асинхронных итераторов и многого другого, что стало возможным после введения ключевых слов `async` и `await`. В результате *Curio* может похвастаться более чистым API и более простой реализацией, чем `asyncio`.

В примере 21.21приведен скрипт *blogdom.py* (пример 21.1), переписанный с использованием *Curio*.

Пример 21.21. blogdom.py: пример 21.1 с использованием Curio

```
#!/usr/bin/env python3
from curio import run, TaskGroup
import curio.socket as socket
from keyword import kwlist

MAX_KEYWORD_LEN = 4

async def probe(domain: str) -> tuple[str, bool]: ❶
    try:
        await socket.getaddrinfo(domain, None) ❷
    except socket.gaierror:
        return (domain, False)
    return (domain, True)

async def main() -> None:
    names = (kw for kw in kwlist if len(kw) <= MAX_KEYWORD_LEN)
    domains = (f'{name}.dev'.lower() for name in names)
    async with TaskGroup() as group: ❸
```

[1] В отличие от JavaScript, где `async` и `await` зашиты во встроенный цикл событий и среду выполнения, т. е. браузер, Node.js или Deno.

```
        for domain in domains:
            await group.spawn(probe, domain) ❹
        async for task in group: ❺
            domain, found = task.result
            mark = '+' if found else ' '
            print(f'{mark} {domain}')

if __name__ == '__main__':
    run(main()) ❻
```

❶ `probe` не нужно получать цикл событий, потому что ...
❷ ... `getaddrinfo` – функция верхнего уровня в модуле `curio.socket`, а не метод объекта `loop`, как в `asyncio`.
❸ Группа задач `TaskGroup` – ключевое понятие в *Curio*, она служит для отслеживания и управления несколькими сопрограммами и гарантирует, что все они выполняются, а по завершении производится очистка.
❹ Метод `TaskGroup.spawn` запускает сопрограмму, управляемую конкретным экземпляром `TaskGroup`. Сопрограмма обернута экземпляром `Task`.
❺ Обход `TaskGroup` с помощью `async for` отдает экземпляры `Task` по мере их завершения. Это соответствует строке примера 21.1, в которой используется конструкция `for … as_completed(…):`.
❻ В *Curio* впервые предложен этот разумный способ запуска асинхронной программы в Python.

Еще пара слов о последнем пункте: в примерах кода на основе `asyncio` в первом издании этой книги снова и снова повторяются такие строки:

```
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

Класс *Curio* `TaskGroup`– это асинхронный контекстный менеджер, который заменяет несколько поспешно придуманных API и способов кодирования в `asyncio`. Мы только что видели, как обход `TaskGroup` делает ненужной функцию `asyncio.as_completed(…)`. Другой пример: следующий фрагмент, взятый из документации по группам задач (*https://curio.readthedocs.io/en/latest/reference.html# task-groups*), применяется вместо специальной функции `gather` для сбора результатов всех задач в группе:

```
async with TaskGroup(wait=all) as g:
    await g.spawn(coro1)
    await g.spawn(coro2)
    await g.spawn(coro3)
print('Results:', g.results)
```

Группы задач поддерживают *структурную конкурентность* (*https://en.wikipedia.org/wiki/Structured_concurrency*): форму конкурентного программирования, в которой вся деятельность группы асинхронных задач ограничена одной точкой входа и одной точкой выхода. Это аналог структурного программирования, которое поставило вне закона команду `GOTO` и ввело блоки, чтобы ограничить количество точек входа и выхода для циклов и подпрограмм. При использовании в качестве асинхронного контекстного менеджера `TaskGroup`

гарантирует, что при выходе из объемлющего блока все задачи, запущенные внутри группы, завершаются или отменяются, а все исключения возбуждаются.

Структурная конкурентность, скорее всего, будет внедрена в `asyncio` в последующих версиях Python. Ясный намек на это появился в документе PEP 654 «Exception Groups and except*» (*https://peps.python.org/pep-0654/*), одобренном для реализации в Python 3.11 (*https://mail.python.org/archives/list/python-dev@python.org/thread/2ORDAW74LGE3ZI2QETPJRT2ZL7MCCPG2/*). В разделе «Мотивация» (*https://peps.python.org/pep-0654/#motivation*) упоминаются «ясли», как в *Trio* называются группы задач: «Реализация улучшенного API запуска задач в *asyncio*, подсказанного яслями Trio, стала основным побудительным мотивом для написания этого PEP».

Еще одна важная особенность *Curio* – улучшенная поддержка программирования с использованием сопрограмм и потоков в одной кодовой базе, что совершенно необходимо в большинстве нетривиальных асинхронных программ. Запуск потока с помощью `await spawn_thread(func, ...)` возвращает объект `AsyncThread` с интерфейсом, похожим на `Task`. Потоки могут вызывать сопрограммы благодаря специальной функции `AWAIT(coro)` – ее имя написано прописными буквами, потому что `await` – теперь ключевое слово.

Curio предлагает также класс `UniversalQueue`, который можно использовать для координации работы, выполняемой потоками, сопрограммами *Curio* и сопрограммами `asyncio`. Все правильно – в *Curio* есть средства, позволяющие работать в потоке и выполнять `asyncio` в другом потоке того же процесса, а взаимодействие при этом осуществляется с помощью `UniversalQueue` и `UniversalEvent`. API этих «универсальных» классов одинаков внутри и вне сопрограмм, но в сопрограмме вызовам необходимо предпосылать `await`.

В октябре 2021 года, когда я пишу эти строки, *HTTPX* стала первой клиентской библиотекой для работы с HTTP, совместимой с *Curio*, но я пока не знаю ни одной асинхронной библиотеки баз данных, поддерживающей ее. В репозитории *Curio* имеется внушительный набор примеров сетевого программирования (*https://github.com/dabeaz/curio/tree/78bca8a6ad677ef51e1568ac7b3e51441ab49c42/examples*), в т. ч. с использованием *WebSocket*, а также реализация конкурентного программирования, описанного в RFC 8305 «Happy Eyeballs» (*https://datatracker.ietf.org/doc/html/rfc8305*), который позволяет подключаться к оконечным точкам IPv6, но быстро переходить к IPv4 при необходимости.

Дизайн *Curio* оказал большое влияние на разработчиков. Каркас *Trio* (*https://trio.readthedocs.io/en/stable/*), начатый Натаниэлем Дж. Смитом, многие идеи заимствует у *Curio*. Возможно также, что *Curio* подтолкнул соразработчиков Python сделать API `asyncio` удобнее для пользователей. Например, в первых версиях пользователям `asyncio` частенько приходилось получать и передавать объект `loop`, потому что некоторые необходимые функции либо были методами `loop`, либо требовали `loop` в качестве аргумента. В недавних версиях Python прямой доступ к циклу событий нужен не так часто; более того, некоторые функции, которые принимали `loop` в качестве факультативного аргумента, теперь объявили эту практику нерекомендуемой.

Следующей нашей темой будут аннотации асинхронных типов.

Аннотации типов для асинхронных объектов

Тип возвращаемого платформенной сопрограммой значения описывает, что мы получим от сопрограммы по завершении `await`, т. е. это тип объекта, который встречается в предложениях `return` в теле платформенной сопрограммы[1].

В этой главе встречалось много примеров аннотированных платформенных сопрограмм, в т. ч. `probe` в примере 21.21:

```
async def probe(domain: str) -> tuple[str, bool]:
    try:
        await socket.getaddrinfo(domain, None)
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

Для аннотирования параметра, который принимает объект сопрограммы, применяется такой обобщенный тип:

```
class typing.Coroutine(Awaitable[V_co], Generic[T_co, T_contra, V_co]):
    ...
```

Этот и следующие типы были введены в версиях Python 3.5 и 3.6 для аннотирования асинхронных объектов:

```
class typing.AsyncContextManager(Generic[T_co]):
    ...
class typing.AsyncIterable(Generic[T_co]):
    ...
class typing.AsyncIterator(AsyncIterable[T_co]):
    ...
class typing.AsyncGenerator(AsyncIterator[T_co], Generic[T_co, T_contra]):
    ...
class typing.Awaitable(Generic[T_co]):
    ...
```

В Python ⩾ 3.9 используйте эквиваленты этих типов из модуля `collections.abc`.

Я хочу подчеркнуть три аспекта этих обобщенных типов.

Первое: все они ковариантны относительно первого параметра-типа, который описывает тип элементов, отдаваемых этими объектами. Вспомните первое из эвристических правил вариантности в главе 15:

> Если формальный параметр-тип определяет тип данных, исходящих из объекта, то он может быть ковариантным.

Второе: `AsyncGenerator` и `Coroutine` контравариантны относительно второго и последующих параметров-типов. Это тип аргумента низкоуровневого метода `.send()`, который цикл событий вызывает для активации асинхронных генераторов и сопрограмм. Поэтому он является «входным» типом, а значит, может быть контравариантным в соответствии со вторым эвристическим правилом вариантности:

---

[1] Это отличается от аннотаций классических сопрограмм, которые мы обсуждали в разделе «Обобщенные типы для классических сопрограмм» главы 17.

> Если формальный параметр-тип определяет тип данных, входящих в объект после его начального конструирования, то он может быть контравариантным.

Третье: `AsyncGenerator` не имеет типа возвращаемого значения, в отличие от `typing.Generator`, который мы видели в разделе «Обобщенные аннотации типов для классических сопрограмм» главы 17. Возврат значения путем возбуждения исключения `StopIteration(value)` был одним из не вполне честных приемов, который позволял генераторам работать как сопрограммы и поддерживать предложение `yield from`, как было показано в разделе «Классические сопрограммы» главы 17. Между асинхронными объектами такого перекрытия нет: объекты `AsyncGenerator` не возвращают значений и полностью отделены от объектов платформенных сопрограмм, которые аннотируются типом `typing.Coroutine`.

Наконец, вкратце рассмотрим преимущества и проблемы, свойственные асинхронному программированию.

Как работает и как не работает асинхронность

В заключительных разделах этой главы обсуждаются высокоуровневые идеи, относящиеся к асинхронному программированию, вне зависимости от используемого языка или библиотеки.

Начнем с объяснения главной причины, по которой асинхронное программирование так привлекательно, а затем развенчаем один популярный миф.

Круги, разбегающиеся вокруг блокирующих вызовов

Райан Дал, автор Node.js, начинает разговор о философии своего проекта словами «Наш подход к вводу-выводу совершенно неверен»[1]. Он определяет *блокирующую функцию* как функцию, выполняющую файловый или сетевой ввод-вывод, и утверждает, что мы не можем обращаться с ними так же, как с неблокирующими функциями. Объясняя причину, он ссылается на числа во втором столбце табл. 21.1.

Таблица 21.1. Характерные для современных компьютеров задержки чтения данных с различных устройств; в третьем столбце данные представлены в масштабе, который проще воспринять человеку

Устройство	Такты CPU	«Человеческий» масштаб
Кеш L1	3	3 секунды
Кеш L2	14	14 секунд
ЗУПВ	250	250 секунд
Диск	41 000 000	1,3 года
Сеть	240 000 000	7,6 лет

Чтобы по достоинству оценить данные в табл. 21.1, имейте в виду, что современные процессоры с тактовой частотой порядка гигагерц выполняют миллиарды тактов в секунду. Предположим, что CPU выполняет 1 миллиард тактов

[1] Видео «Introduction to Node.js» (*https://www.youtube.com/watch?v=M-sc73Y-zQA*), отметка 4:55.

в секунду. Тогда за 1 секунду он может произвести 333 миллиона чтений из кеша L1 или 4 (четыре!) чтения из сети. В третьем столбце эти числа приведены к другому масштабу путем умножения на постоянный коэффициент. Таким образом, в этой альтернативной вселенной чтение из кеша L1 занимает 3 секунды, а из сети 7,6 лет!

Таблица 21.1 объясняет, почему дисциплинированный подход к асинхронному программированию может резко увеличить производительность серверов. Проблема в том, как добиться соблюдения этой дисциплины. Первый шаг – осознать, что «системы, ограниченные вводом-выводом», – фантазия.

Миф о системах, ограниченных вводом-выводом

В разговорах часто повторяют, что асинхронное программирование хорошо для «систем, ограниченных вводом-выводом». Из собственного опыта я вынес урок, что не бывает таких систем. Ограничены вводом-выводом могут быть *функции*. Быть может, подавляющее большинство функций в вашей системе ограничено вводом-выводом, т. е. они тратят больше времени на ожидание ввода-вывода, чем на обработку данных. А пока ждут, уступают управление циклу событий, который может активировать другие готовые к выполнению задачи. Но в любой нетривиальной системе неизбежно есть счетные части, ограниченные производительностью процессора. И даже в тривиальных системах такие места обнаруживаются при работе под нагрузкой. В разделе «Поговорим» ниже я расскажу историю о двух асинхронных программах, которые сражались со счетными функциями, замедлявшими цикл событий до такой степени, что это оказывало серьезное влияние на производительность.

Учитывая, что в любой нетривиальной системе есть счетные функции, ключом к успеху асинхронного программирования является выработка правильного подхода к ним.

Как не попасть в ловушку счетных функций

Если вы пишете крупные программы на Python, то, вероятно, имеете автоматизированные регрессионные тесты, предназначенные для своевременного обнаружения падения производительности. Это критически важно для асинхронного кода, но относится и к многопоточному – из-за GIL. Если дождаться момента, когда замедление начнет тревожить команду разработчиков, то будет уже слишком поздно. Для исправления, скорее всего, понадобится масштабная переделка.

Предложу несколько вариантов действий при обнаружении пожирателей процессорного времени:

- делегировать задачу пулу Python-процессов;
- делегировать задачу внешней очереди задач;
- переписать часть кода на Cython, C, Rust или другом языке, который компилируется в машинный код и имеет интерфейс к Python/C API, предпочтительно с освобождением GIL;
- решить, что вы можете позволить себе падение производительности и ничего не делать, но запротоколировать это решение, чтобы было проще вернуться к нему позже.

Решение о выборе внешней очереди задачи и интеграции с ней следует принимать в самом начале проекта, чтобы все члены команды без колебаний использовали ее в случае необходимости.

Последний вариант – ничего не делать – попадает в категорию технического долга (*https://en.wikipedia.org/wiki/Technical_debt*).

Конкурентное программирование – чрезвычайно увлекательная тема, и я мог бы написать еще много чего. Но она не является центральной для этой книги, а эта глава и так уже получилась одной из самых длинных, так что я закругляюсь.

Резюме

> Проблема всех обычных подходов к асинхронному программированию в том, что они предлагают все или ничего. Вам придется переписать весь код, чтобы нигде ничего не блокировалось, иначе вы просто зря потратите время.
>
> – Альваро Видела и Джейсон Дж. У. Уильямс, «RabbitMQ in Action»

Я выбрал этот эпиграф по двум причинам. На верхнем уровне он напоминает нам избегать блокирования цикла событий, делегируя медленные задачи другой единице выполнения – от простого потока до распределенной очереди задач. На нижнем уровне он предупреждает: стоит написать первое `async def`, как в программе неизбежно будут появляться все новые и новые `async def`, `await`, `async with` и `async for`. И внезапно использование неасинхронных библиотек становится проблемой.

После простых примеров вращающегося индикатора в главе 19 мы здесь сосредоточились на асинхронном программировании с применением платформенных сопрограмм. Мы начали со скрипта проверки доменных имен *blogdom.py*, а затем познакомились с понятием *объектов, допускающих ожидание*. Читая исходный код скрипта *flags_asyncio.py*, мы встретились с первым примером *асинхронного контекстного менеджера*.

Продолжая исследование вариантов программы загрузки флагов, мы ввели в рассмотрение две мощные функции: генератор `asyncio.as_completed` и сопрограмму `loop.run_in_executor`. Мы также познакомились с идеей семафора и применили его, чтобы ограничить количество конкурентных операций загрузки – такого поведения ожидают от добропорядочных HTTP-клиентов.

Серверное асинхронное программирование было представлено примерами *mojifinder*: на основе веб-службы *FastAPI* и скрипта *tcp_mojifinder.py* – в последнем использовалась только библиотека `asyncio` и протокол TCP.

Асинхронные итераторы и итерируемые объекты стали следующей крупной темой, которой посвящены разделы о конструкции `async for`, асинхронной консоли Python, асинхронных генераторах, асинхронных генераторных выражениях и асинхронных включениях.

Последний пример в этой главе – скрипт *blogdom.py*, переписанный с применением каркаса *Curio* с целью продемонстрировать, что асинхронные средства Python не привязаны к пакету `asyncio`. *Curio* также иллюстрирует концепцию *структурной конкурентности*, которая, возможно, будет принята всей индустрией и позволит сделать конкурентный код понятнее.

Наконец, в разделе «Как работает и как не работает асинхронность» и его подразделах мы обсудили, чем привлекает асинхронное программирование, развечали миф о «системах, ограниченных вводом-выводом», и дали рекомендации о том, что делать с неизбежными в любой программе счетными частями.

Дополнительная литература

Основной доклад Дэвида Бизли на конференции PyOhio 2016 «Fear and Awaiting in Async» (*https://www.youtube.com/watch?v=E-1Y4kSsAFc*) – фантастическое, приправленное живым кодом введение в потенциал языковых средств, открывшийся в результате добавления Юрием Селивановым ключевых слов `async` и `await` в версию Python 3.5. В одном месте Бизли сетует, что `await` нельзя использовать в списковых включениях, но эта проблема была решена Селивановым в документе PEP 530 «Asynchronous Comprehensions» (*https://peps.python.org/pep-0530/*) и реализована в Python 3.6 в том же году. Если не считать этого момента, все остальное в докладе Бизли не привязано ко времени, поскольку он демонстрирует, как асинхронные объекты, описанные в этой главе, работают вообще без поддержки каркаса – нужна лишь простая функция `run`, которая приводит в действие сопрограммы с помощью вызова `.send(None)`. Только в самом конце Бизли представляет проект *Curio* (*https://github.com/dabeaz/curio*), который он начал в том же году в качестве эксперимента, чтобы посмотреть, насколько далеко можно зайти в асинхронном программировании, не имея фундамента в виде обратных вызовов и будущих объектов, а пользуясь только сопрограммами. Как выяснилось, зайти можно очень далеко, и это демонстрирует эволюция *Curio* и последующее создание *Trio* (*https://trio.readthedocs.io/en/stable/*) Натаниэлем Дж. Смитом. В документации по *Curio* есть дополнительные ссылки на выступления Бизли по этому предмету.

Помимо *Trio* Натаниэл Дж. Смит написал две глубокие статьи в блоге, которые я горячо рекомендую: «Some thoughts on asynchronous API design in a post-async/await world» (*https://vorpus.org/blog/some-thoughts-on-asynchronous-api-design-in-a-post-asyncawait-world/*), в которой сравнивается дизайн *Curio* и *asyncio*, и «Notes on structured concurrency, or: Go statement considered harmful» (*https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/*) о структурной конкурентности. Смит также дал развернутый и весьма информативный ответ на вопрос «What is the core difference between asyncio and trio?» (*https://stackoverlow.com/questions/4948-2969/what-is-the-core-difference-between-asyncio-and-trio*) на сайте StackOverflow.

Для желающих узнать больше о пакете *asyncio* я уже упоминал лучшие известные мне на момент написания этой главы ресурсы: официальная документация (*https://docs.python.org/3/library/asyncio.html*) после знаменательной переделки (*https://bugs.python.org/issue33649*), предпринятой Юрием Селивановым в 2018 году, и книга Калеба Хэттинга «Using Asyncio in Python» (O'Reilly). В официальной документации обязательно прочитайте раздел «Разработка с применением asyncio» (*https://docs.python.org/3/library/asyncio-dev.html*), где описан отладочный режим *asyncio*, а также обсуждаются типичные ошибки и подводные камни и как их избежать.

Вполне доступное 30-минутное введение в асинхронное программирование вообще и с применением *asyncio* в частности имеется в ролике Мигуэля Гринберга «Asynchronous Python for the Complete Beginner» (*https://www.youtube.com/watch?v=iG6fr81xHKA*), представленном на конференции PyCon 2017. Из еще одного замечательного введения, «Demystifying Python's Async and Await Keywords» (*https://www.youtube.com/watch?v=F19R_M4Nay4*), представленного Майклом Кеннеди, я среди прочего узнал о библиотеке *unsync* (*https://asherman.io/projects/unsync.html*), которая предоставляет декоратор для делегирования выполнения сопрограмм, функций, ограниченных вводом-выводом, и счетных функций модулям `asyncio`, `threading` или `multiprocessing` – в зависимости от потребности.

На конференции EuroPython 2019 Линн Рут, возглавляющая глобальное сообщество *PyLadies*, выступила с блестящей презентацией «Advanced asyncio: Solving Real-world Production Problems» (*https://m.youtube.com/watch?v=sW76-pRkZk8*), в основу которой лег ее опыт использования Python во время работ инженером в штате компании Spotify.

В 2020 году Лукаш Ланга записал серию видео, посвященных *asyncio*, начинающуюся с «Learn Python's AsyncIO #1 – The Async Ecosystem» (*https://www.youtube.com/watch?v=XbI7XjFYsN4*). Ланга также сделал превосходный ролик «AsyncIO + Music» (*https://www.youtube.com/watch?v=02CLD-42VdI*) для конференции PyCon 2020, в котором не только показано применение библиотеки *asyncio* к очень конкретной событийно-ориентированной предметной области, но и объясняется вся ее архитектура снизу доверху.

Еще одна область, в которой событийно-ориентированное программирование занимает ведущие позиции, – встраиваемые системы. Именно поэтому Дамьен Джордж добавил поддержку `async/await` в свой интерпретатор *MicroPython* (*https://micropython.org/*) для микроконтроллеров. На конференции PyCon Australia 2018 Мэтт Трентини продемонстрировал библиотеку *uasyncio* (*https://docs.micropython.org/en/latest/library/uasyncio.html*), подмножество *asyncio*, которая вошла в стандартную библиотеку MicroPython.

Если вам интересны рассуждения об асинхронном программировании в Python на верхнем уровне, ознакомьтесь со статьей «Python async frameworks – Beyond developer tribalism» (*https://www.encode.io/articles/python-async-frameworks-beyond-developer-tribalism*) в блоге Тома Кристи.

Наконец, я рекомендую статью Боба Нистрёма «What Color Is Your Function?», в которой обсуждаются несовместимые модели выполнения обычных и асинхронных функций (сопрограмм) в JavaScript, Python, C# и других языках. Спойлер: Нистрём приходит к выводу, что правильно все сделано в языке Go, где все функции одного цвета. Мне нравится эта особенность Go. Но я также думаю, что у Натаниэля Дж. Смита были резоны для написания статьи «Go statement considered harmful» (*https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/*). Ничто в мире не совершенно, а конкурентное программирование – вообще штука сложная.

Поговорим

Как медленная функция чуть не подпортила тесты производительности uvloop

В 2016 году Юрий Селиванов выпустил *uvloop* (*https://github.com/MagicStack/uvloop*), «быструю совместимую по интерфейсу замену циклу событий *asyncio*». Тесты производительности, представленные Селивановым в статье, анонсирующей библиотеку, очень впечатляли. Он писал: «Она по крайней мере в 2 раза быстрее nodejs, gevent и многих других асинхронных каркасов на Python. Производительность asyncio на основе uvloop близка к производительности программ на Go».

Однако автор не скрывает, что *uvloop* может сравняться с производительностью Go при двух условиях:

1. Go сконфигурирован с одним потоком. В результате среда выполнения Go ведет себя аналогично *asyncio*: конкурентность достигается за счет нескольких сопрограмм, управляемых циклом событий, и все они работают в одном потоке[1].
2. В коде Python 3.5 используется не только *uvloop*, но и *httptools* (*https://github.com/MagicStack/httptools*).

Селиванов объясняет, что написал *httptools* после тестирования производительности *uvloop* в сочетании с *aiohttp* (*https://docs.aiohttp.org/en/stable/*) – одной из первых полнофункциональных библиотек HTTP, построенных на основе `asyncio`:

> Однако причиной низкой производительности *aiohttp* оказался входящий в нее синтаксический анализатор HTTP, настолько медленный, что уже не важно, насколько быстро работает базовая библиотека ввода-вывода. Ради интереса мы написали интерфейс к Python для библиотеки *http-parser* (написанная на C библиотека разбора HTTP для Node.js, первоначально разработанная для *NGINX*). Библиотека называется *httptools* и доступна на Github и PyPI.

А теперь подумайте вот о чем: тесты производительности HTTP, разработанные Селивановым, включали простой эхо-сервер, написанный на разных языках с разными библиотеками, который нагружался с помощью инструмента эталонного тестирования *wrk* (*https://github.com/wg/wrk*). Большинство разработчиков назвали бы простой эхо-сервер «системой, ограниченной вводом-выводом», так? Но оказалось, что разбор HTTP-заголовков ограничен быстродействием процессора, а его реализация в *aiohttp* работала медленно, когда Селиванов прогонял свои тесты производительности в 2016 году. Как только написанная на Python функция начинала разбирать заголовки, цикл событий блокировался. Это влияние было настолько значительным, что Селиванов взял на себя дополнительный труд по написанию *httptools*. Без оптимизации счетного кода весь выигрыш в производительности, достигаемый за счет более быстрого цикла событий, терялся.

[1] Однопоточный режим подразумевался по умолчанию до выхода версии Go 1.5. А в предшествующие годы Go заслуженно заработал репутацию прекрасного языка для высококонкурентных сетевых систем. Еще одно свидетельство в пользу того, что для конкурентности не нужно ни несколько потоков, ни несколько процессорных ядер.

Смерть от тысячи ран

Представьте, что вместо простого эхо-сервера имеется сложная развивающаяся система на Python, насчитывающая тысячи строк асинхронного кода и зависящая от многих внешних библиотек. Много лет назад меня попросили диагностировать проблемы производительности в такой системе. Она была написана на Python 2.7 с применением библиотеки *Twisted* (*https://twistedmatrix.com/trac/*), во многих отношениях предшественницы самой `asyncio`.

Python служил для построения фасада к пользовательскому веб-интерфейсу, интегрируя функциональность уже имевшихся библиотек и инструментов командной строки, написанных на других языках, но не предназначавшихся для конкурентного выполнения.

Проект был амбициозным; он разрабатывался уже больше года, но до сих пор не дошел до стадии эксплуатации[1]. Со временем разработчики начали замечать, что производительность системы в целом снижается, и никак не могли найти, где затык.

А произошло вот что: с каждой добавленной функцией новый счетный код замедлял цикл событий *Twisted*. Роль Python как связующего языка означала, что нужно разбивать данные и преобразовывать их из одного формата в другой – и таких операций было много. Не было какого-то одного узкого места: проблема расползлась по бессчетному количеству небольших функций, которые добавлялись в течение всего времени разработки. Чтобы исправить ситуацию, пришлось бы переосмыслить архитектуру системы, переписать массу кода, быть может, воспользоваться очередью задач, микросервисами или библиотеками, написанными на языках, более приспособленных к конкурентному решению счетных задач. Инвесторы были не готовы вкладывать дополнительные деньги, и проект вскоре был свернут.

Когда я рассказал эту историю Глифу Лефковицу, основателю проекта *Twisted*, он сказал, что один из его приоритетов в начале проекта, включающего асинхронное программирование, – решить, какими инструментами пользоваться для вынесения счетных задач из цикла событий. Слова Глифа стали побудительным мотивом для написания раздела «Как не попасть в ловушку счетных функций» выше.

[1] Независимо от технических решений, это, наверное, была самая большая ошибка в проекте: заинтересованные стороны проигнорировали подход MVP – подготовить минимально работоспособный продукт (Minimum Viable Product) как можно скорее, а затем в постоянном темпе добавлять новые функции.

Часть V

Метапрограммирование

Глава 22

Динамические атрибуты и свойства

> Ценность свойств заключается в том, что благодаря им можно совершенно безопасно – и это даже рекомендуется – раскрывать атрибуты-данные как часть открытого интерфейса класса.
>
> – Мартелли, Равенскрофт, Холден, «Почему свойства важны»[1]

Атрибуты-данные и методы в Python носят общее название «атрибуты»; метод – это просто *вызываемый* атрибут. Помимо атрибутов-данных и методов, мы можем создавать еще свойства, позволяющие заменить открытые атрибуты-данные методами-акцессорами (т. е. методами чтения и установки), не изменяя интерфейс класса. Это согласуется с *принципом единообразного доступа*:

> Все сервисы, предоставляемые модулем, должны быть доступны с помощью единообразной нотации, скрывающей механизм реализации: хранение или вычисление[2].

В Python есть несколько способов реализовать динамические атрибуты. В этой главе рассматриваются самые простые: декоратор `@property` и специальный метод `__getattr__`.

Пользовательский класс, в котором имеется метод `__getattr__`, может реализовать вариант динамических атрибутов, который я называю *виртуальными атрибутами*; они не объявлены в исходном коде класса и отсутствуют в экземпляре `__dict__`, но могут быть получены из какого-то другого места или вычислены «на лету», когда программа пытается прочитать несуществующий атрибут, например `obj.no_such_attr`.

Динамические атрибуты – вид метапрограммирования, обычно применяемый авторами каркасов. Однако в Python базовая техника настолько проста, что любой человек может воспользоваться ими, даже для повседневных задач обработки данных. С нее мы и начнем эту главу.

Что нового в этой главе

Побудительным мотивом для большей части изменений в этой главе стало обсуждение декоратора `@functools.cached_property` (появившегося в Python 3.8) и ис-

---

[1] *Alex Martelli, Anna Ravenscroft, Steve Holden.* Python in a Nutshell. 3-е изд. O'Reilly. С. 123.

[2] *Bertrand Meyer.* Object-Oriented Software Construction. 2-е изд. С. 57.

пользование `@property` в сочетании с `@functools.cache` (новшество в 3.9). Это повлияло на код классов `Record` и `Event` в разделе «Вычисляемые свойства». Я также переработал код, применив оптимизацию, описанную в документе PEP 412 «Key-Sharing Dictionary» (*https://peps.python.org/pep-0412/*).

Чтобы уделить больше внимания наиболее существенным средствам и сохранить удобочитаемость кода, я убрал менее важный код: объединил старый класс `DbRecord` с `Record`, заменил `shelve.Shelve` на `dict` и удалил логику скачивания набора данных OSCON – теперь примеры читаются из локального файла, входящего в репозиторий кода на сопроводительном сайте (*https://github.com/fluentpython/example-code-2e*).

Применение динамических атрибутов для обработки данных

В примерах ниже мы воспользуемся динамическими атрибутами для обработки данных в формате JSON, опубликованных издательством O'Reilly для конференции OSCON 2014. В примере 19.1 показаны четыре записи из этого набора[1].

Пример 22.1. Примеры записей из файла osconfeed.json; значения некоторых полей сокращены

```
{ «Schedule»:
  { "conferences": [{"serial": 115 }],
    "events": [
      { "serial": 34505,
        "name": "Why Schools Don/t Use Open Source to Teach Programming",
        "event_type": "40-minute conference session",
        "time_start": "2014-07-23 11:30:00",
        "time_stop": "2014-07-23 12:10:00",
        "venue_serial": 1462,
        "description": "Aside from the fact that high school programming...",
        "website_url": "http://oscon.com/oscon2014/public/schedule/detail/34505",
        "speakers": [157509],
        "categories": ["Education"] }
    ],
    "speakers": [
      { "serial": 157509,
        "name": "Robert Lefkowitz",
        "photo": null,
        "url": "http://sharewave.com/",
        "position": "CTO",
        "affiliation": "Sharewave",
        "twitter": "sharewaveteam",
        "bio": "Robert /r0ml/ Lefkowitz is the CTO at Sharewave, a startup..." }
    ],
```

[1] Конференция OSCON – O'Reilly Open Source Conference – пала жертвой пандемии COVID-19. Оригинального JSON-файла размером 744 КБ, который я использовал для этих примеров, по состоянию на 10 января 2021 года в сети не было. Его копию под названием *osconfeed.json* можно найти в репозитории кода к этой книге (*https://github.com/fluentpython/example-code-2e/blob/master/22-dyn-attr-prop/oscon/data/osconfeed.json*).

```
    "venues": [
      { "serial": 1462,
        "name": "F151",
        "category": "Conference Venues" }
    ]
  }
}
```

В примере 22.1 показаны четыре из 895 записей JSON-файла. Как видим, весь набор данных – это единственный JSON-объект с ключом «Schedule», значением которого является отображение с четырьмя ключами: «conferences» (конференции), «events» (мероприятия), «speakers» (докладчики) и «venues» (места проведения). С каждым из четырех ключей ассоциирован список записей. В полном наборе данных списки «events», «speakers» и «venues» содержат десятки и даже сотни записей, тогда как список «conferences» состоит всего из одной записи, показанной выше. В каждой записи имеется поле «serial», уникально идентифицирующее запись в пределах списка.

Для исследования этого набора данных я воспользовался консолью Python, как показано в примере 22.2.

Пример 22.2. Интерактивное изучение osconfeed.json

```
>>> import json
>>> with open('data/osconfeed.json') as fp:
...     feed = json.load(fp) ❶
>>> sorted(feed['Schedule'].keys()) ❷
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed['Schedule'].items()):
...     print(f'{len(value):3} {key}') ❸
...
1 conferences
484 events
357 speakers
53 venues
>>> feed['Schedule']['speakers'][-1]['name'] ❹
'Carina C. Zona'
>>> feed['Schedule']['speakers'][-1]['serial'] ❺
141590
>>> feed['Schedule']['events'][40]['name']
'There *Will* Be Bugs'
>>> feed['Schedule']['events'][40]['speakers'] ❻
[3471, 5199]
```

❶ Загрузить как словарь `dict`, содержащий вложенные словари и списки со строковыми и целыми значениями.
❷ Вывести все четыре коллекции записей внутри «Schedule».
❸ Вывести счетчики записей в каждой коллекции.
❹ Обойти вложенные словари и списки, чтобы получить имя последнего докладчика.
❺ Получить порядковый номер этого докладчика.
❻ У каждого мероприятия есть список `'speakers'`, содержащий нуль или более порядковых номеров докладчиков.

Исследование JSON-подобных данных с динамическими атрибутами

Пример 22.2 достаточно прост, но синтаксис `feed['Schedule']['events'][40]['name']` слишком громоздкий. В JavaScript то же самое можно было бы записать в виде `feed.Schedule.events[40].name`. На Python нетрудно реализовать похожий на словарь класс, который ведет себя подобным образом, – в сети нет недостатка в примерах[1]. Я написал класс `FrozenJSON`, который проще большинства готовых, т. к. поддерживает только чтение; он предназначен исключительно для исследования данных. Однако он рекурсивный и автоматически обрабатывает вложенные отображения и списки.

В примере 22.3 демонстрируется использование класса `FrozenJSON`, а в примере 22.4 приведен его исходный код.

Пример 22.3. Класс `FrozenJSON` из примера 22.4 позволяет читать атрибуты, например `name`, и вызывать методы, например `.keys()` и `.items()`

```
>>> import json
>>> raw_feed = json.load(open('data/osconfeed.json'))
>>> feed = FrozenJSON(raw_feed) ❶
>>> len(feed.Schedule.speakers) ❷
357
>>> feed.keys()
dict_keys(['Schedule'])
>>> sorted(feed.Schedule.keys()) ❸
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed.Schedule.items()): ❹
...     print(f'{len(value):3} {key}')
...
1 conferences
484 events
357 speakers
53 venues
>>> feed.Schedule.speakers[-1].name ❺
'Carina C. Zona'
>>> talk = feed.Schedule.events[40]
>>> type(talk) ❻
<class 'explore0.FrozenJSON'>
>>> talk.name
'There *Will* Be Bugs'
>>> talk.speakers ❼
[3471, 5199]
>>> talk.flavor ❽
Traceback (most recent call last):
  ...
KeyError: 'flavor'
```

❶ Построить экземпляр `FrozenJSON` по словарю `raw_feed`, содержащему вложенные словари и списки.

❷ `FrozenJSON` допускает обход вложенных словарей с помощью нотации атрибутов; здесь мы получаем длину списка докладчиков.

[1] Два примера: AttrDict (*https://pypi.python.org/pypi/attrdict*) и addict (*https://pypi.python.org/pypi/addict*).

❸ Методы скрытых за объектом `FrozenJSON` словарей также доступны, например метод `.keys()` возвращает имена коллекций.
❹ С помощью метода `items()` мы можем извлечь имена коллекций записей и их содержимое, чтобы показать длину каждого значения.
❺ Список, например `feed.Schedule.speakers`, остается списком, но те объекты внутри него, которые являются отображениями, преобразуются в тип `FrozenJSON`.
❻ Элемент 40 списка `events` был объектом типа JSON; теперь это экземпляр класса `FrozenJSON`.
❼ С каждым мероприятием связан список `speakers`, содержащий порядковые номера докладчиков.
❽ При попытке прочитать несуществующий атрибут возбуждается исключение `KeyError`, а не `AttributeError`, как обычно.

Краеугольным камнем класса `FrozenJSON` является метод `__getattr__`, которым мы уже пользовались в примере класса `Vector` из раздела «Vector, попытка № 3: доступ к динамическим атрибутам» главы 12, чтобы обращаться к компонентам вектора по буквам – `v.x`, `v.y`, `v.z` и т. д. Напомним, что интерпретатор вызывает специальный метод `__getattr__`, только если обычный поиск атрибута завершается неудачно (т. е. именованный атрибут не удается найти ни в экземпляре, ни в классе, ни в его суперклассах).

Последняя строка в примере 22.3 выявляет небольшой дефект моего кода: попытка чтения несуществующего атрибута должна бы возбуждать исключение `AttributeError`, а не `KeyError`, как у меня. Я даже реализовал такую обработку ошибок, но при этом метод `__getattr__` стал вдвое длиннее, и это отвлекало внимание от той важной логики, которую я стремился продемонстрировать. Учитывая, что пользователи, вероятно, знают, что `FrozenJSON` состоит из отображений и списков, я полагаю, что ошибка `KeyError` никого не смутит.

Пример 22.4. explore0.py: преобразование набора данных из формата JSON в объект `FrozenJSON`, содержащий вложенные объекты `FrozenJSON`, списки и значения примитивных типов

```
from collections import abc
class FrozenJSON:
    """Допускающий только чтение фасад для навигации по JSON-подобному
       объекту с применением нотации атрибутов
    """

    def __init__(self, mapping):
        self.__data = dict(mapping) ❶

    def __getattr__(self, name): ❷
        try:
            return getattr(self.__data, name) ❸
        except AttributeError:
            return FrozenJSON.build(self.__data[name]) ❹

    def __dir__(self): ❺
        return self.__data.keys()

    @classmethod
```

```
    def build(cls, obj): ❻
        if isinstance(obj, abc.Mapping): ❼
            return cls(obj)
        elif isinstance(obj, abc.MutableSequence): ❽
            return [cls.build(item) for item in obj]
        else:
            return obj
```

❶ Построить объект `dict` по аргументу `mapping`. Тем самым мы проверяем, что получили словарь (или нечто, что можно преобразовать в словарь). Два знака подчеркивания в начале `__data` говорят, что это *закрытый атрибут*.

❷ Метод `__getattr__` вызывается, только когда не существует атрибута с именем `name`.

❸ Если имени `name` соответствует какой-то атрибут словаря `__data`, возвращаем его. Так обрабатываются вызовы методов типа `feed.keys()`: метод `keys` является атрибутом словаря `__data`.

❹ В противном случае получаем элемент с ключом `name` из `self.__data` и возвращаем результат вызова для него метода `FrozenJSON.build()`[1].

❺ Это альтернативный конструктор, типичное применение декоратора `@classmethod`.

❻ Если `obj` – отображение, строим по нему объект `FrozenJSON`. Это пример *гусиной типизации* (смотрите раздел «Гусиная типизация» главы 13, если подзабыли).

❼ Если это экземпляр `MutableSequence`, то он должен быть списком[2], поэтому строим список, рекурсивно передавая каждый элемент `obj` методу `.build()`.

❽ Если это не `dict` и не `list`, возвращаем элемент без изменения.

Экземпляр `FrozenJSON` имеет закрытый атрибут экземпляра `__data`, хранящийся под именем `_FrozenJSON__data`, как было объяснено в разделе «Закрытые и “защищенные” атрибуты в Python» главы 11. Попытка получить атрибут с любым другим именем приводит к вызову `__getattr__`. Этот метод сначала смотрит, есть ли в словаре `self.__data` атрибут (не ключ!) с таким именем; это позволяет экземплярам `FrozenJSON` обрабатывать методы самого класса `dict`, например `items`, делегируя работу методу `self.__data.items()`. Если в `self.__data` нет атрибута с именем `name`, то `__getattr__` использует `name` как ключ, читает из `self.__dict` элемент с таким ключом и передает его методу `FrozenJSON.build`. Это позволяет обходить вложенные структуры в JSON-данных, поскольку каждое вложенное отображение преобразуется в новый экземпляр `FrozenJSON` методом класса `build`.

Отметим, что исходный набор данных не кешируется и не трансформируется. При его обходе вложенные структуры данных всякий раз преобразуются заново в тип `FrozenJSON`. Но при таком размере набора это приемлемо, да и наш скрипт предназначен только для исследования и преобразования данных.

---

[1] Именно в этой строке может возникнуть исключение `KeyError`: в выражении `self.__data[name]`. Его следует обработать и подменить исключением `AttributeError`, поскольку такого исключения вызывающая программа ожидает от `__getattr__`. Прилежному читателю предлагается написать этот код в качестве упражнения.

[2] Источником данных является объект типа JSON, а он поддерживает только два типа коллекций: `dict` и `list`.

Любой скрипт, который генерирует или эмулирует динамические атрибуты с именами, полученными из произвольного источника, должен помнить об одной проблеме: ключи, хранящиеся в исходных данных, могут не удовлетворять правилам образования имен атрибутов. В следующем разделе мы займемся этой проблемой.

Проблема недопустимого имени атрибута

У класса `FrozenJSON` есть ограничение: в нем не предусмотрена специальная обработка имен атрибутов, являющихся ключевыми словами Python. Например, построив объект вида:

```
>>> student = FrozenJSON({'name': 'Jim Bo', 'class': 1982})
```

мы не сможем прочитать атрибут `grad.class`, т. к. `class` – зарезервированное слово в Python:

```
>>> student.class
  File "<stdin>", line 1
    student.class
          ^
SyntaxError: invalid syntax
```

Конечно, можно сделать так:

```
>>> getattr(student, 'class')
1982
```

Но идея класса `FrozenJSON` заключалась в том, чтобы предоставить удобный доступ к данным, поэтому лучше проверять, является ли ключ отображения, переданного методу `FrozenJSON.__init__`, зарезервированным словом, и если да, то добавлять в конец символ `_`, чтобы атрибут можно было прочитать так:

```
>>> student.class_
1982
```

Для этого достаточно заменить однострочный метод `__init__` из примера 22.4 кодом, показанным ниже.

Пример 22.5. explore1.py: добавление `_` в имена атрибутов, являющиеся зарезервированными словами Python

```
    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if keyword.iskeyword(key): ❶
                key += '_'
            self.__data[key] = value
```

❶ Функция `keyword.iskeyword(…)` – именно то, что нам нужно; для ее использования необходимо импортировать модуль `keyword`.

Похожая проблема может возникнуть, если ключ в JSON-данных не является допустимым идентификатором Python:

```
>>> x = FrozenJSON({'2be':'or not'})
>>> x.2be
  File "<stdin>", line 1
    x.2be
      ^
SyntaxError: invalid syntax
```

Такие проблематичные ключи легко выявить в Python 3, где класс `str` предоставляет метод `s.isidentifier()`, который сообщает, является ли `s` допустимым идентификатором с точки зрения грамматики языка Python. Но преобразование ключа, не являющегося допустимым идентификатором, в допустимое имя атрибута – нетривиальная задача. Возможное решение – реализовать метод `__getitem__`, чтобы разрешить доступ к атрибуту с помощью нотации вида `x['2be']`. Для простоты я проигнорирую этот случай.

Уделив внимание именам динамических атрибутов, обратимся к другой важной особенности класса `FrozenJSON`: логике метода класса `build`. `FrozenJSON.build` вызывается из `__getattr__` для получения объектов разных типов в зависимости от значения обрабатываемого атрибута: вложенные структуры преобразуются в экземпляры `FrozenJSON` или списки экземпляров `FrozenJSON`.

Как мы увидим ниже, ту же логику можно было бы реализовать не в методе класса, а в специальном методе `__new__`.

Гибкое создание объектов с помощью метода __new__

Мы часто называем `__init__` конструктором, но это только потому, что позаимствовали терминологию из других языков. В Python метод `__init__` получает `self` в качестве первого аргумента, поэтому к моменту вызова `__init__` интерпретатором объект уже существует. Кроме того, `__init__` не может ничего возвращать. Так что в действительности это инициализатор, а не конструктор.

Когда класс вызывается для создания экземпляра, Python вызывает специальный метод класса `__new__`. Хотя это метод класса, обрабатывается он не так, как другие: к нему не применяется декоратор `@classmethod`. Python принимает экземпляр, возвращенный `__new__`, и передает его в качестве первого аргумента `self` методу `__init__`. Мы редко пишем `__new__` самостоятельно, потому что реализации, унаследованной от `object`, обычно достаточно.

При необходимости метод `__new__` может вернуть экземпляр другого класса. В таком случае интерпретатор не вызывает `__init__`. Иными словами, логика создания объекта в Python описывается следующим псевдокодом:

```
# псевдокод конструирования объекта
def make(the_class, some_arg):
    new_object = the_class.__new__(some_arg)
    if isinstance(new_object, the_class):
        the_class.__init__(new_object, some_arg)
    return new_object

# следующие предложения приблизительно эквивалентны
x = Foo('bar')
x = make(Foo, 'bar')
```

В примере 22.6 показан вариант класса `FrozenJSON`, в котором логика метода класса `build` перенесена в метод `__new__`.

Пример 22.6. explore2.py: использование `__new__` вместо `build` для конструирования новых объектов, которые могут быть или не быть экземплярами `FrozenJSON`

```
from collections import abc
import keyword

class FrozenJSON:
    """Допускающий только чтение фасад для навигации по JSON-подобному
       объекту с применением нотации атрибутов
    """

    def __new__(cls, arg): ❶
        if isinstance(arg, abc.Mapping):
            return super().__new__(cls) ❷
        elif isinstance(arg, abc.MutableSequence): ❸
            return [cls(item) for item in arg]
        else:
            return arg

    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if keyword.iskeyword(key):
                key += '_'
            self.__data[key] = value

    def __getattr__(self, name):
        try:
            return getattr(self.__data, name)
        except AttributeError:
            return FrozenJSON(self.__data[name]) ❹

    def __dir__(self):
        return self.__data.keys()
```

❶ Будучи методом класса, `__new__` получает в качестве первого аргумента сам класс, а остальные аргументы – те же, что получает `__init__`, за исключением `self`.

❷ По умолчанию работа делегируется методу `__new__` суперкласса. В данном случае мы вызываем метод `__new__` из базового класса `object`, передавая ему `FrozenJSON` в качестве единственного аргумента.

❸ Оставшаяся часть `__new__` ничем не отличается от прежнего метода `build`.

❹ Здесь раньше вызывался метод `FrozenJSON.build`, а теперь мы просто вызываем класс `FrozenJSON`, а Python обрабатывает это как вызов `FrozenJSON.__new__`.

Метод `__new__` получает в качестве первого аргумента класс, потому что обычно создается экземпляр именно этого класса. Таким образом, при вызове `super().__new__(cls)` из `FrozenJSON.__new__` в действительности вызывается `object.__new__(FrozenJSON)`, а объект, построенный классом `object`, является экземпляром класса `FrozenJSON`. Атрибут `__class__` нового экземпляра содержит ссыл-

ку на `FrozenJSON`, хотя собственно конструирование производилось методом `object.__new__`, реализованным на C в недрах интерпретатора.

Структура набора данных OSCON не очень пригодна для интерактивного исследования. Например, для мероприятия с индексом `40`, озаглавленного `'There *Will* Be Bugs'`, зарегистрировано два докладчика, `3471` и `5199`, но найти их нелегко, потому что это порядковые номера, а не индексы в списке `Schedule.speakers`. Чтобы получить запись о докладчике, придется выполнить линейный поиск по списку, пока не отыщется подходящий порядковый номер. Наша следующая задача – изменить структуру данных и автоматизировать извлечение связанных записей.

Вычисляемые свойства

С декоратором `@property` мы впервые познакомились в главе 11, в разделе «Хешируемый класс Vector2d». В примере 11.7 я использовал два свойства `Vector2d` просто для того, чтобы сделать атрибуты `x` и `y` доступными только для чтения. Здесь мы рассмотрим свойства, которые вычисляют значения, и это подведет нас к вопросу о том, как кешировать такие значения.

Записи в списке `'events'` JSON-данных OSCON содержат последовательные целые числа, указывающие на записи в списках `'speakers'` и `'venues'`. Например, ниже показана запись о выступлении на конференции (описание сокращено):

```
{ "serial": 33950,
  "name": "There *Will* Be Bugs",
  "event_type": "40-minute conference session",
  "time_start": "2014-07-23 14:30:00",
  "time_stop": "2014-07-23 15:10:00",
  "venue_serial": 1449,
  "description": "If you're pushing the envelope of programming...",
  "website_url": "http://oscon.com/oscon2014/public/schedule/detail/33950",
  "speakers": [3471, 5199],
  "categories": ["Python"] }
```

Мы реализуем класс `Event` со свойствами `venue` и `speakers`, которые будут автоматически возвращать связанные данные, т. е. «разыменовывать» порядковый номер. В примере 22.7 показано желаемое поведение для заданного экземпляра `Event`.

Пример 22.7. Чтение `venue` и `speakers` возвращает объекты `Record`

```
>>> event ❶
<Event 'There *Will* Be Bugs'>
>>> event.venue ❷
<Record serial=1449>
>>> event.venue.name ❸
'Portland 251'
>>> for spkr in event.speakers: ❹
...     print(f'{spkr.serial}: {spkr.name}')
...
3471: Anna Martelli Ravenscroft
5199: Alex Martelli
```

❶ Если имеется экземпляр `Event` …
❷ … то чтение `event.venue` возвращает объект `Record` вместо порядкового номера.
❸ Теперь легко получить название `venue`.
❹ Свойство `event.speakers` возвращает список экземпляров `Record`.

Как обычно, будем разрабатывать код по шагам и начнем с класса `Record` и функции, которая читает JSON-данные и возвращает `dict`, содержащий экземпляры `Record`.

Шаг 1: создание управляемого данными атрибута

В примере 22.8 показан тест, которым мы будем руководствоваться на первом шаге.

Пример 22.8. Тест для разработки schedule_v1.py (из примера 22.9)

```
>>> records = load(JSON_PATH) ❶
>>> speaker = records['speaker.3471'] ❷
>>> speaker ❸
<Record serial=3471>
>>> speaker.name, speaker.twitter ❹
('Anna Martelli Ravenscroft', 'annaraven')
```

❶ Загрузить в `dict` JSON-данные функцией `load`.
❷ Ключами `records` являются строки, образованные типом записи и порядковым номером.
❸ `speaker` – экземпляр класса `Record`, определенного в примере 22.9.
❹ Поля первичных JSON-данных можно получить в виде атрибутов экземпляра `Record`.

Код скрипта *schedule_v1.py* приведен в примере 22.9.

Пример 22.9. schedule_v1.py: реорганизация данных о мероприятиях OSCON

```
import json

JSON_PATH = 'data/osconfeed.json'

class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs) ❶

    def __repr__(self):
        return f'<{self.__class__.__name__} serial={self.serial!r}>' ❷

    def load(path=JSON_PATH):
        records = {} ❸
        with open(path) as fp:
            raw_data = json.load(fp) ❹
        for collection, raw_records in raw_data['Schedule'].items(): ❺
            record_type = collection[:-1] ❻
            for raw_record in raw_records:
                key = f'{record_type}.{raw_record["serial"]}' ❼
                records[key] = Record(**raw_record) ❽
        return records
```

❶ Стандартная идиома для построения экземпляра, атрибуты которого создаются из именованных аргументов (подробности см. ниже).
❷ Использовать поле `serial`, чтобы построить представление `Record`, показанное в примере 22.8.
❸ Метод `load` в конечном итоге вернет словарь экземпляров `Record`.
❹ Разобрать JSON и вернуть объекты Python: списки, словари, числа и т. д.
❺ Обойти все четыре списка верхнего уровня: `'conferences'`, `'events'`, `'speakers'` и `'venues'`.
❻ `record_type` – имя списка без последнего символа, т. е. `speakers` становится `speaker`. В Python ⩾ 3.9 то же самое можно сделать явно, написав `collection.removesuffix('s')` – см. документ PEP 616 «String methods to remove prefixes and suffixes» (*https://peps.python.org/pep-0616/*).
❼ Построить ключ в формате `'speaker.3471'`.
❽ Создать экземпляр `Record` и сохранить его в словаре `records` под ключом `key`.

В методе `Record.__init__` иллюстрируется распространенный при программировании на Python прием. Напомню, что в словаре `__dict__` объекта хранятся атрибуты – если только в классе не объявлен атрибут `__slots__` (см. раздел «Экономия памяти с помощью атрибута класса __slots__» главы 11). Поэтому копирование в `__dict__` отображения – быстрый способ создать сразу несколько атрибутов (bunch of attributes) экземпляра[1].

В зависимости от приложения в классе `Record`, возможно, придется иметь дело с ключами, которые не являются допустимыми именами атрибутов, как мы видели в разделе «Проблема недопустимого имени атрибута» выше. Обсуждение этой проблемы отвлекло бы нас от главной идеи данного примера, да и в наборе данных, который мы читаем, такой проблемы нет.

Определение класса `Record` в примере 22.9 настолько простое, что вы, наверное, недоумеваете, почему мы не использовали его раньше вместо более сложного класса `FrozenJSON`. Причины две. Во-первых, `FrozenJSON` рекурсивно преобразует вложенные отображения и списки; в классе `Record` это не нужно, потому что в преобразованном наборе данных нет отображений, вложенных в другие отображения или списки. Записи могут содержать только строки, целые числа, списки строк и списки целых чисел. Вторая причина: `FrozenJSON` предоставляет доступ к внутреннему словарю атрибутов `__data`, – который мы использовали для вызова методов, например `keys`, – а здесь эта функциональность не нужна.

В стандартной библиотеке Python есть классы, аналогичные нашему классу `Record`, экземпляры которых содержат произвольный набор атрибутов, переданных `__init__` в виде именованных аргументов: `types.SimpleNamespace` (*https://docs.python.org/3/library/types.html#types.SimpleNamespace*), `argparse.Namespace` (*https://docs.python.org/3/library/argparse.html#argparse.Namespace*)

[1] Кстати, `Bunch` – имя класса, который Алекс Мартелли в 2001 году использовал при публикации этого рецепта, названного им «The simple but handy collector of a bunch of named stuff class» (*https://github.com/ActiveState/code/tree/master/recipes/Python/52308_simple_but_handy_collector_bunch_named_stuff*).

> и `multiprocessing.Namespace` (*https://docs.python.org/3/library/multiprocessing.html#multiprocessing.managers.Namespace*). Я написал более простой класс `Record`, чтобы проиллюстрировать существо этой идеи: обновление атрибута `__dict__` экземпляра в методе `__init__`.

После проделанной реорганизации набора данных мы можем расширить класс `Record`, так чтобы он автоматически выбирал записи `venue` и `speaker`, на которые ссылается запись `event`. Для реализации этой идеи мы воспользуемся свойствами.

Шаг 2: выборка связанных записей с помощью свойств

Цель следующей версии такова: пусть имеется запись `event`, тогда чтение ее атрибута `venue` должно возвращать `Record`. Примерно то же самое делает Django ORM, когда мы обращаемся к полю `models.ForeignKey`: вместо ключа мы получаем связанный объект модели.

Начнем со свойства `venue`. В примере 22.10 показана часть консольного сеанса.

Пример 22.10. Извлечение из тестов для файла schedule_v2.py

```
>>> event = Record.fetch('event.33950') ❶
>>> event ❷
<Event 'There *Will* Be Bugs'>
>>> event.venue ❸
<Record serial=1449>
>>> event.venue.name ❹
'Portland 251'
>>> event.venue_serial ❺
1449
```

❶ Статический метод `Record.fetch` извлекает экземпляр `Record` или `Event` из набора данных.
❷ Отметим, что `event` – экземпляр класса `Event`.
❸ Доступ к атрибуту `event.venue` возвращает экземпляр `Record`.
❹ Теперь легко найти название места проведения `event.venue`.
❺ У экземпляра `Event` также имеется атрибут `venue_serial`, прочитанный из JSON-данных.

`Event` – подкласс `Record`, в который добавлено поле `venue` для получения связанных записей и специализированный метод `__repr__`.

Приведенный в этом разделе код находится в модуле *schedule2.py* в репозитории кода к этой книге (*https://github.com/fluentpython/example-code-2e/blob/master/22-dyn-attr-prop/oscon/schedule_v2.py*). Поскольку код занимает около 60 строк, я разобью его на части и начну с улучшенного класса `Record`.

Пример 22.11. schedule_v2.py: класс `Record` с новым методом `fetch`

```
import inspect ❶
import json

JSON_PATH = 'data/osconfeed.json'
```

```
class Record:

    __index = None ❷

    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        return f'<{self.__class__.__name__} serial={self.serial!r}>'

    @staticmethod ❸
    def fetch(key):
        if Record.__index is None: ❹
            Record.__index = load()
        return Record.__index[key] ❺
```

❶ `inspect` будет вызываться из метода `load`, показанного в примере 22.13.
❷ В закрытом атрибуте класса `__index` будет храниться ссылка на `dict`, возвращенный методом `load`.
❸ `fetch` сделан статическим методом, чтобы было понятно, что его действие не зависит от экземпляра или класса, от имени которого он вызывается.
❹ Заполнить `Record.__index`, если необходимо.
❺ Нужно, чтобы извлечь запись с заданным ключом `key`.

Это пример, когда использование декоратора `staticmethod` имеет смысл. Метод `fetch` всегда применяется к атрибуту класса `Record.__index`, даже если вызван из подкласса, как, например, в случае `Event.fetch()`, который мы рассмотрим ниже. Сделав его методом класса, мы только запутали бы читателя, потому что первый аргумент `cls` не использовался бы.

Теперь рассмотрим использование свойства в классе `Event` (пример 22.12).

Пример 22.12. schedule_v2.py: класс `Event`

```
class Event(Record): ❶

    def __repr__(self):
        try:
            return f'<{self.__class__.__name__} {self.name!r}>' ❷
        except AttributeError:
            return super().__repr__()

    @property
    def venue(self):
        key = f'venue.{self.venue_serial}'
        return self.__class__.fetch(key) ❸
```

❶ Класс `Event` расширяет `Record`.
❷ Если в экземпляре есть атрибут `name`, включаем его в строковое представление. В противном случае делегируем методу `__repr__`, унаследованному от `Record`.
❸ Свойство `venue` строит ключ `key` по атрибуту `venue_serial` и передает его методу класса `fetch`, унаследованному от `Record` (зачем используется `self.__class__`, объяснено ниже).

Во второй строке метода `venue` в примере 22.12 возвращается `self.__class__.fetch(key)`. Почему бы не написать просто `self.fetch(key)`? Это более простое выражение работает для набора данных OSCON, потому что в нем нет записи о мероприятии с ключом `'fetch'`. Но если бы в записи о мероприятии такой ключ был, то в соответствующем экземпляре `Event` выражение `self.fetch` было бы значением этого поля, а не ссылкой на метод класса `fetch`, унаследованный классом `Event` от `Record`. Это тонкая ошибка, которая легко могла бы остаться незамеченной при тестировании, потому что зависит от набора данных.

При создании имен атрибутов экземпляра из данных всегда существует риск ошибок вследствие маскирования атрибутов класса (например, методов) или потери данных из-за случайного перезаписывания уже существующих атрибутов экземпляра. Эта опасность является, пожалуй, основной причиной, по которой словари в Python по умолчанию не похожи на объекты JavaScript.

Если бы класс `Record` больше походил на отображение, т. е. реализовывал динамический метод `__getitem__`, а не `__gettarr__`, то можно было бы не опасаться ошибок, вызванных маскированием или перезаписью. Пользовательское отображение, наверное, является наиболее отвечающим духу Python способом реализации `Record`. Но если бы я пошел по этому пути, то у нас не было бы шанса поразмышлять о ловушках, подстерегающих нас при программировании динамических атрибутов.

И последняя часть примера – переделанная функция `load`.

Пример 22.13. schedule2.py: функция `load`

```
def load(path=JSON_PATH):
    records = {}
    with open(path) as fp:
        raw_data = json.load(fp)
    for collection, raw_records in raw_data['Schedule'].items():
        record_type = collection[:-1] ❶
        cls_name = record_type.capitalize() ❷
        cls = globals().get(cls_name, Record) ❸
        if inspect.isclass(cls) and issubclass(cls, Record): ❹
            factory = cls ❺
        else:
            factory = Record ❻
        for raw_record in raw_records: ❼
            key = f'{record_type}.{raw_record["serial"]}'
            records[key] = factory(**raw_record) ❽
    return records
```

❶ До сих пор нет отличий от функции `load` из файла schedule_*v1.py* (пример 22.9).

❷ Преобразовать первую букву `record_type` в верхний регистр, чтобы получить потенциальное имя класса (например, `'event'` превращается в `'Event'`).

❸ Получить объект с таким именем из глобальной области видимости модуля; если такого объекта нет, получаем `Record`.

❹ Если только что полученный объект – класс, который является подклассом `Record`, то …

❺ … связать с ним имя `factory`. Это означает, что `factory` может быть произвольным подклассом `Record`, определяемым переменной `record_type`.
❻ В противном случае связать имя `factory` с `Record`.
❼ Цикл `for`, в котором создаются ключи и сохраняются записи, такой же, как и раньше, с тем исключением, что…
❽ … объект, сохраняемый в `records`, конструируется функцией `factory`, которая может быть конструктором класса `Record` или его подкласса – в зависимости от значения `record_type`.

Отметим, что единственное значение `record_type`, для которого существует пользовательский класс, – это `Event`, но если бы мы написали классы с именами `Speaker` или `Venue`, то `load` автоматически использовала бы при построении и сохранении записей их, а не подразумеваемый по умолчанию класс `Record`.

Теперь применим ту же идею к новому свойству `speakers` в классе `Events`.

Шаг 3: переопределение существующего атрибута свойством

Имя свойства `venue` в примере 22.12 не совпадает с именем поля в записях коллекции `«events»`. Данные для него берутся из поля с именем `venue_serial`. С другой стороны, в каждой записи коллекции `events` имеется поле `speakers`, содержащее список порядковых номеров. Мы хотим раскрыть эту информацию в виде свойства `speakers` экземпляра `Event`, которое возвращало бы список экземпляров `Record`. Этот конфликт имен требует особого внимания, как следует из примера 22.14.

Пример 22.14. schedule_v3.py: свойство `speakers`

```
@property
def speakers(self):
    spkr_serials = self.__dict__['speakers'] ❶
    fetch = self.__class__.fetch
    return [fetch(f'speaker.{key}')
            for key in spkr_serials] ❷
```

❶ Нужные нам данные находятся в атрибуте `speakers`, но получить их мы должны непосредственно из экземпляра `__dict__`, чтобы избежать рекурсивного обращения к свойству `speakers`.
❷ Вернуть список всех записей с ключами, соответствующими числам в `spkr_serials`.

Внутри метода `speakers` попытка прочитать `self.speakers` вызовет само свойство, что очень быстро приведет к исключению `RecursionError`. Однако если читать те же самые данные из `self.__dict__['speakers']`, то мы обойдем обычный алгоритм Python, предназначенный для получения атрибутов, свойство не будет вызвано и мы избежим рекурсии. По этой причине чтение или запись напрямую в атрибут объекта `__dict__` является общепринятой практикой метапрограммирования в Python.

При вычислении `obj.my_attr` интерпретатор сначала смотрит на класс `obj`. Если в классе имеется свойство с именем `my_attr`, то оно маскирует одноименный атрибут экземпляра. Это будет продемонстрировано на примерах из раздела «Свойства переопределяют атрибуты экземпляра» ниже, а в главе 23 мы узнаем, что свойство реализовано как дескриптор – более мощная и общая абстракция.

Когда я кодировал списковое включение в примере 22.14, в мой пещерный программистский мозг закралась мысль: «Это может оказаться дорого». На самом деле нет, потому что на мероприятиях OSCON докладчиков немного, поэтому написание более сложного кода может оказаться преждевременной оптимизацией. Однако потребность в кешировании свойств возникает часто, а вместе с ней и подводные камни. Поэтому в следующих примерах мы посмотрим, как это делается.

Шаг 4: кеширование свойств на заказ

Кеширование свойств – востребованная операция, потому что пользователь естественно ожидает, что вычисление выражения вида `event.venue` не должно быть дорогим[1]. Какая-то форма кеширования может оказаться необходимой, если метод `Record.fetch`, стоящий за свойствами класса `Event`, обращается с запросом к базе данных или веб API.

В первом издании книги я кодировал пользовательскую логику кеширования для метода `speakers`, как показано в примере 22.15.

Пример 22.15. Пользовательская логика кеширования с применением `hasattr` отключает оптимизацию разделения ключей

```
@property
def speakers(self):
    if not hasattr(self, '__speaker_objs'): ❶
        spkr_serials = self.__dict__['speakers']
        fetch = self.__class__.fetch
        self.__speaker_objs = [fetch(f'speaker.{key}')
                for key in spkr_serials]
    return self.__speaker_objs ❷
```

❶ Если в экземпляре нет атрибута с именем `__speaker_objs`, выбрать объекты докладчиков и сохранить их здесь.

❷ Вернуть `self.__speaker_objs`.

Самодельное кеширование в примере 22.15 прямолинейно, но создание атрибута после инициализации экземпляра отменяет оптимизацию, описанную в документе PEP 412 «Key-Sharing Dictionary» (*https://peps.python.org/pep-0412/*), как объясняется в разделе «Практические последствия внутреннего устройства класса dict» главы 3. В зависимости от размера набора данных разница в потреблении памяти может оказаться существенной.

[1] На самом деле это оборотная сторона принципа единообразного доступа Мейера, который я упоминал во вступлении к этой главе. Если вам интересна дискуссия на эту тему, почитайте врезку «Поговорим» в конце главы.

Похожее самодельное решение, не противоречащее оптимизации разделения ключей, требует написания метода `__init__` в классе `Event`, который создаст атрибут `__speaker_objs` и инициализирует его значением `None`. Затем этот атрибут можно будет проверить в методе `speakers`. См. пример 22.16.

Пример 22.16. Память выделяется в `__init__`, чтобы не подавлять оптимизацию разделения ключей

```
class Event(Record):

    def __init__(self, **kwargs):
        self.__speaker_objs = None
        super().__init__(**kwargs)

# 15 строк опущено...
    @property
    def speakers(self):
        if self.__speaker_objs is None:
            spkr_serials = self.__dict__['speakers']
            fetch = self.__class__.fetch
            self.__speaker_objs = [fetch(f'speaker.{key}')
                    for key in spkr_serials]
      return self.__speaker_objs
```

В примерах 22.15 и 22.16 иллюстрируется простая техника кеширования, часто встречающаяся в унаследованных кодовых базах на Python. Однако в многопоточных программах подобные самодельные кеши приводят к состоянию гонки и потенциальному повреждению данных. Если два потока читают свойство, которое не было ранее кешировано, то первый поток должен будет вычислить данные, хранящиеся в кешированном атрибуте (`__speaker_objs` в наших примерах), а второй поток может прочитать еще не полностью сформированное кеширование значение.

К счастью, в Python 3.8 появился потокобезопасный декоратор `@functools.cached_property`. Но, к сожалению, вместе с ним пришли и две «засады», которые объясняются ниже.

Шаг 5: кеширование свойств с помощью functools

Модуль `functools` предлагает три декоратора для кеширования. С `@cache` и `@lru_cache` мы познакомились в разделе «Запоминание с помощью functools.cache» главы 9. А в версии Python 3.8 появился `@cached_property`.

Декоратор `functools.cached_property` кеширует результат метода в одноименном атрибуте экземпляра. Например, в примере 22.17 значение, вычисленное методом `venue`, хранится в атрибуте `venue` в `self`. Когда впоследствии клиент попытается прочитать `venue`, будет использован атрибут `venue`, а не метод.

Пример 22.17. Простое использование `@cached_property`

```
    @cached_property
    def venue(self):
        key = f'venue.{self.venue_serial}'
        return self.__class__.fetch(key)
```

В разделе «Шаг 3: переопределение существующего атрибута свойством» выше мы видели, что свойство маскирует атрибут экземпляра с тем же именем. Но как тогда может работать `@cached_property`? Коль скоро свойство переопределяет атрибут экземпляра, атрибут `venue` будет проигнорирован и всегда будет вызываться метод `venue`, который будет каждый раз вычислять `key` и обращаться к `fetch`!

Ответ немного удручает: имя `cached_property` неправильное. Декоратор `@cached_property` не создает полноценного свойства, он создает *непереопределяющий дескриптор*. Дескриптор – это объект, который управляет доступом к атрибуту в другом классе. Мы будем подробно изучать дескрипторы в главе 23. Декоратор `property` – это высокоуровневый API для создания *переопределяющего дескриптора*. В главе 23 объясняется разница между тем и другим.

А пока отложим в сторону реализацию и сосредоточимся на различиях между `cached_property` и `property` с точки зрения пользователя. Раймонд Хэттингер очень хорошо объяснил это в документации по Python (*https://docs.python.org/3/library/functools.html#functools.cached_property*):

> Механизмы работы `cached_property()` и `property`() различаются. Обычное свойство запрещает запись в атрибут, если не определен метод установки. А `cached_property` разрешает запись всегда.
>
> Декоратор `cached_property` работает только в момент проверки существования и только при условии, что атрибута с таким же именем не существует.
>
> В этом случае `cached_property` записывает в атрибут с таким же именем. Последующие операции чтения и записи имеют более высокий приоритет, чем метод `cached_property`, поэтому кешированное свойство работает, как обычный атрибут.
>
> Кешированное значение можно очистить, удалив атрибут. После этого метод `cached_property` будет работать снова[1].

Но вернемся к классу `Event`: специальное поведение `@cached_property` делает его непригодным для декорирования `speakers`, потому что этот метод предполагает существование атрибута с таким же именем `speakers`, в котором хранятся порядковые номера докладчиков на мероприятии.

Декоратор `@cached_property` имеет несколько важных ограничений:

- его нельзя использовать в качестве замены `@property`, если декорируемый метод уже зависит от существования одноименного атрибута экземпляра;
- его нельзя использовать в классе, где определен атрибут `__slots__`;
- он подавляет оптимизацию разделения ключей в экземпляре `__dict__`, потому что создает атрибут экземпляра после `__init__`.

[1] Источник: документация по `@functools.cached_property` (*https://docs.python.org/3/library/functools.html#functools.cached_property*). Я знаю, что автором этого объяснения является Раймонд Хэттингер, потому что он прислал мне его в ответ на зарегистрированную мной проблему: bpo42781 – в документации по functools.cached_property должно быть объяснено, что это не переопределяющий дескриптор (*https://bugs.python.org/issue42781*). Хэттингер – один из основных авторов официальной документации по Python и стандартной библиотеки. Он также написал блестящее руководство по дескрипторам «Descriptor HowTo Guide» (*https://docs.python.org/3/howto/descriptor.html*), которое стало основным источником для главы 23.

Несмотря на эти ограничения, `@cached_property` решает распространенную проблему простым способом и является потокобезопасным. Его Python-код (*https://github.com/python/cpython/blob/e6d0107e13ed957109e79b796984d3d026a8660d/Lib/functools.py#L926*) дает пример использования *реентерабельной блокировки* (*https://docs.python.org/3/library/threading.html#rlock-objects*).

В документации по `@cached_property` (*https://docs.python.org/3.10/library/ functools.html#functools.cached_property*) рекомендуется альтернативное решение, которое можно применить к методу `speakers`: образовать композицию декораторов `@property` и `@cache`, как показано в примере 22.18.

Пример 22.18. Композиция `@property` и `@cache`

```
@property ❶
@cache ❷
def speakers(self):
    spkr_serials = self.__dict__['speakers']
    fetch = self.__class__.fetch
    return [fetch(f'speaker.{key}')
            for key in spkr_serials]
```

❶ Порядок важен: `@property` должен быть расположен выше…
❷ … `@cache`.

Вспомните семантику этой синтаксической конструкции (врезка «Композиция декораторов» в главе 9). Первые три строки примера 22.18 аналогичны следующей записи:

```
speakers = property(cache(speakers))
```

Декоратор `@cache` применяется к методу `speakers` и возвращает новую функцию. Затем к этой функции применяется декоратор `@property`, который заменяет ее вновь сконструированным свойством.

На этом мы завершаем обсуждение свойств, допускающих только чтение, и кеширующих декораторов, а вместе с ним и исследование набора данных OSCON. В следующем разделе начинается новая серия примеров, в которых создаются свойства, допускающие чтение и запись.

Использование свойств для контроля атрибутов

Помимо вычисления значений атрибутов, свойства применяются с целью реализации бизнес-правил. Для этого открытый атрибут заменяется атрибутом, защищенным методами чтения и установки, без изменения клиентского кода. Рассмотрим развернутый пример.

LineItem, попытка № 1: класс строки заказа

Представим себе приложение для магазина, который продает натуральные пищевые продукты вразвес, т. е. клиенты могут заказывать орехи, сухофрукты или хлопья по весу. В такой системе заказ состоит из последовательности строк, а каждую строку можно представить классом, показанным в примере 22.19.

Пример 22.19. bulkfood_v1.py: простейший класс `LineItem`

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

Красиво и просто. Пожалуй, слишком просто. В примере 20.20 показано, в чем проблема.

Пример 20.20. Если вес отрицателен, то и промежуточный итог отрицателен

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> raisins.subtotal()
69.5
>>> raisins.weight = -20 # мусор на входе...
>>> raisins.subtotal()   # мусор на выходе...
-139.0
```

Пример, конечно, несерьезный, но не такой надуманный, как можно было бы предположить. Вот правдивая история, случившаяся, когда сайт Amazon.com только зарождался:

> Мы обнаружили, что покупатель мог заказать отрицательное количество книг! И мы бы перечислили на его кредитную карту соответствующую сумму и, надо полагать, ждали бы, когда он отгрузит книги.
>
> – Джефф Безос, основатель и генеральный директор Amazon.com[1]

Как это исправить? Можно было бы изменить интерфейс класса `LineItem`, добавив методы чтения и установки атрибута `weight`. Так поступают в Java, и ничего плохого в этом нет.

С другой стороны, было бы естественно устанавливать атрибут `weight` элемента заказа, просто присваивая ему значение, да и не исключено, что в других частях эксплуатируемой системы уже встречается прямой доступ к атрибуту вида `item.weight`. В таком случае следовало бы заменить атрибут-данные свойством – это было бы в духе Python.

LineItem, попытка № 2: контролирующее свойство

Реализовав свойство, мы сможем использовать методы чтения и установки, но интерфейс класса `LineItem` при этом не изменится (т. е. для установки атрибута `weight` объекта `LineItem` по-прежнему нужно будет написать `raisins.weight = 12`).

В примере 22.21 приведен код свойства `weight`, допускающего чтение и запись.

[1] Цитата из статьи «Рождение продавца» Джеффа Безоса в журнале «Уолл-стрит джорнал» (*http://on.wsj.com/1ECl8Dl*) (15 октября 2011). Обратите внимание, что по состоянию на 2021 год для чтения этой статьи нужно оформить подписку.

Пример 22.21. bulkfood_v2.py: класс `LineItem` со свойством `weight`

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight ❶
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property ❷
    def weight(self): ❸
        return self.__weight ❹

    @weight.setter ❺
    def weight(self, value):
        if value > 0:
            self.__weight = value ❻
        else:
            raise ValueError('value must be > 0') ❼
```

❶ Здесь уже используется метод установки свойства, который гарантирует, что не может быть создан экземпляр с отрицательным значением `weight`.
❷ Декоратором `@property` обозначается метод чтения свойства.
❸ Имена всех методов, реализующих свойство, совпадают с именем открытого атрибута: `weight`.
❹ Фактическое значение хранится в закрытом атрибуте `__weight`.
❺ У декорированного метода чтения свойства имеется атрибут `.setter`, который является также и декоратором; тем самым методы чтения и установки связываются между собой.
❻ Если значение больше нуля, присваиваем его закрытому атрибуту `__weight`.
❼ В противном случае возбуждаем исключение `ValueError`.

Теперь объект `LineItem` с недопустимым весом создать невозможно:

```
>>> walnuts = LineItem('walnuts', 0, 10.00)
Traceback (most recent call last):
  ...
ValueError: value must be > 0
```

Итак, мы защитили атрибут `weight` от присваивания отрицательных значений пользователем. Но хотя покупатели обычно не вправе устанавливать цену товара, в результате ошибки служащего или программы все же может быть создан объект `LineItem` с отрицательной ценой `price`. Чтобы предотвратить и это, мы могли бы преобразовать `price` в свойство, но это повлекло бы за собой частичное повторение кода.

Напомним слова Пола Грэхема, приведенные в главе 17: «Видя в своих программах повторяющиеся структуры, я расцениваю их как знак беды». Лекарство от повторения – абстрагирование. Существует два способа абстрагировать определения свойств: фабрика свойств и дескрипторный класс. Подход на основе дескрипторного класса обладает большей гибкостью, мы посвятим ему всю главу 20. На самом деле сами свойства реализованы как дескрипторные

классы. А пока продолжим наше исследование и реализуем фабрику свойств в виде функции.

Но прежде необходимо лучше понять природу свойств.

Правильный взгляд на свойства

Встроенная функция `property` часто используется как декоратор, но в действительности она является классом. В Python функции и классы нередко взаимозаменяемы, поскольку являются вызываемыми объектами и не существует оператора `new` для создания объекта, поэтому вызов конструктора ничем не отличается от вызова фабричной функции. Как функцию, так и класс можно использовать в качестве декоратора, при условии что они возвращают новый вызываемый объект, являющийся подходящей заменой декорированной функции.

Вот полная сигнатура конструктора класса `property`:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Все аргументы необязательны; если для какого-то из них не указана функция, то результирующий объект свойства не поддерживает соответствующую операцию.

Тип `property` появился в версии Python 2.2, но синтаксис декоратора был добавлен только в версии Python 2.4, т. е. на протяжении нескольких лет свойства нужно было определять, передавая функции-акцессоры в первых двух аргументах.

«Классический» синтаксис определения свойств без декораторов показан в примере 22.22.

Пример 22.22. bulkfood_v2b.py: то же, что пример 22.21, но без декораторов

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self): ❶
        return self.weight * self.price

    def get_weight(self):
        return self.__weight

    def set_weight(self, value): ❷
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
        weight = property(get_weight, set_weight) ❸
```

❶ Простой метод чтения.
❷ Простой метод установки.
❸ Построить свойство и присвоить его открытому атрибуту класса.

В некоторых случаях классическая форма удобнее синтаксиса декораторов, одним из примеров является код фабрики свойств, который мы вскоре обсудим. С другой стороны, в теле класса, где много методов, декораторы позволяют сразу опознать методы чтения и установки, не полагаясь на соглашение о префиксах `get` и `set` в именах.

Наличие свойств в классе влияет на то, как можно искать атрибуты в экземплярах такого класса, и, на первый взгляд, это удивительно. Объясним, в чем здесь дело.

Свойства переопределяют атрибуты экземпляра

Свойства всегда являются атрибутами класса, но на самом деле они управляют доступом к атрибутам в экземплярах этого класса.

В разделе «Переопределение атрибутов класса» главы 11 мы видели, что если экземпляр и его класс оба имеют атрибут-данные с одним и тем же именем, то атрибут экземпляра переопределяет, или маскирует, атрибут класса – по крайней мере, когда мы обращаемся к атрибуту от имени этого экземпляра. Проблема демонстрируется в примере 22.23.

Пример 22.23. Атрибут экземпляра маскирует атрибут-данные класса

```
>>> class Class: ❶
...     data = 'the class data attr'
...     @property
...     def prop(self):
...         return 'the prop value'
...
>>> obj = Class()
>>> vars(obj) ❷
{}
>>> obj.data ❸
'the class data attr'
>>> obj.data = 'bar' ❹
>>> vars(obj) ❺
{'data': 'bar'}
>>> obj.data ❻
'bar'
>>> Class.data ❼
'the class data attr'
```

❶ Определить `Class` с двумя атрибутами класса: атрибутом-данными `data` и свойством `prop`.

❷ `vars` возвращает атрибут `__dict__` объекта `obj`; как видим, атрибутов экземпляра в нем нет.

❸ Чтение из `obj.data` возвращает значение `Class.data`.

❹ Запись в `obj.data` создает атрибут экземпляра.

❺ Проинспектировать экземпляр, чтобы узнать, какие у него атрибуты.

❻ Теперь, читая `obj.data`, мы получаем значение атрибута экземпляра. При чтении из экземпляра `obj` атрибут экземпляра `data` маскирует атрибут класса `data`.

❼ Атрибут `Class.data` не изменился.

Попробуем теперь переопределить атрибут `prop` экземпляра `obj`. В примере 22.24 показано продолжение предыдущего сеанса.

Пример 22.24. Атрибут экземпляра не маскирует свойство класса (продолжение примера 22.23)

```
>>> Class.prop ❶
<property object at 0x1072b7408>
>>> obj.prop ❷
'the prop value'
>>> obj.prop = 'foo' ❸
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
>>> obj.__dict__['prop'] = 'foo' ❹
>>> vars(obj) ❺
{'data': 'bar', 'prop': 'foo'}
>>> obj.prop ❻
'the prop value'
>>> Class.prop = 'baz' ❼
>>> obj.prop ❽
'foo'
```

❶ Чтение `prop` непосредственно из `Class` возвращает сам объект свойства, при этом его метод чтения не выполняется.
❷ Чтение `obj.prop` приводит к выполнению метода чтения.
❸ Попытка установить атрибут экземпляра `prop` завершается ошибкой.
❹ Запись `'prop'` напрямую в `obj.__dict__` работает.
❺ Как видим, теперь у `obj` есть два атрибута экземпляра: `data` и `prop`.
❻ Однако при чтении `obj.prop` по-прежнему выполняется метод чтения свойства. Свойство не маскируется атрибутом экземпляра.
❼ В случае перезаписывания `Class.prop` объект свойства уничтожается.
❽ Теперь чтение `obj.prop` возвращает атрибут экземпляра. `Class.prop` больше не является свойством, поэтому и не переопределяет `obj.prop`.

В качестве заключительной демонстрации добавим новое свойство в `Class` и убедимся, что оно переопределяет атрибут экземпляра. Пример 22.25 продолжает предыдущий.

Пример 22.25. Новое свойство класса маскирует существующий атрибут экземпляра (продолжение примера 22.24)

```
>>> obj.data ❶
'bar'
>>> Class.data ❷
'the class data attr'
>>> Class.data = property(lambda self: 'the "data" prop value') ❸
>>> obj.data ❹
'the "data" prop value'
>>> del Class.data ❺
>>> obj.data ❻
'bar'
```

❶ `obj.data` возвращает атрибут экземпляра `data`.
❷ `Class.data` возвращает атрибут класса `data`.
❸ Перезаписать `Class.data` новым свойством.
❹ Теперь `Class.data` маскирует `obj.data`.
❺ Удалить свойство.
❻ Теперь `obj.data` снова возвращает атрибут экземпляра `data`.

В этом разделе мы прежде всего хотели показать, что при вычислении выражения вида `obj.data` поиск `data` начинается не с `obj`. На самом деле поиск начинается с `obj.__class__`, и только если в классе не существует свойства с именем `data`, то Python заглядывает в сам объект `obj`. Это правило применимо к *переопределяющим дескрипторам* вообще, а свойства являются лишь их частным случаем. Мы отложим дальнейшее рассмотрение дескрипторов до главы 23.

А пока вернемся к свойствам. В любой единице кода Python – модулях, функциях, классах, методах – может присутствовать строка документации. В следующем разделе мы увидим, как строка документации присоединяется к свойствам.

Документирование свойств

Когда функции оболочки `help()` или интегрированной среды разработки нужно вывести документацию по свойству, она получает информацию из атрибута свойства `__doc__`.

В случае классического синтаксиса конструктор класса `property` может получить строку документации в виде аргумента `doc`:

```
weight = property(get_weight, set_weight, doc='weight in kilograms')
```

Строка документации метода чтения – того, который снабжен декоратором `@property`, – становится документацией свойства в целом. На рис. 22.1 показано, как выглядят строки документации для кода из примера 22.26.

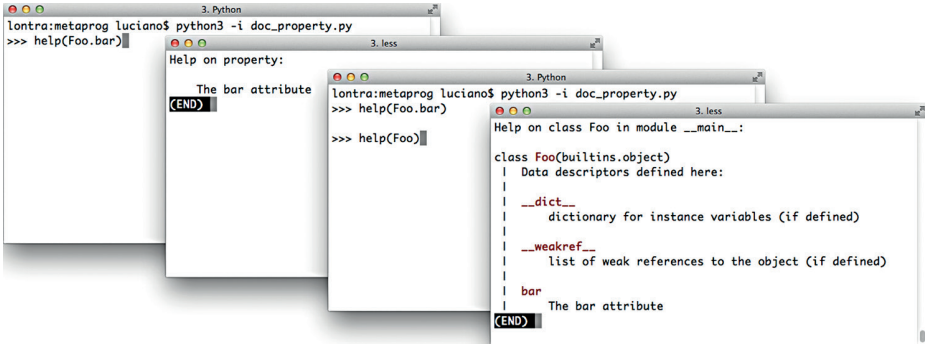


Рис. 22.1. Как выглядит оболочка Python после выполнения команд `help(Foo.bar)` и `help(Foo)`. Исходный код см. в примере 22.26

Пример 22.26. Документирование свойства

```
class Foo:

    @property
    def bar(self):
        """Атрибут bar """
        return self.__dict__['bar']

    @bar.setter
    def bar(self, value):
        self.__dict__['bar'] = value
```

Разобравшись с основами, вернемся к вопросу о том, как защитить атрибуты `weight` и `price` экземпляра `LineItem`, чтобы им можно было присвоить только положительные значения, – но при этом не писать вручную две почти одинаковые пары методов чтения и установки.

Программирование фабрики свойств

Мы создадим фабрику свойств `quantity` – такое название выбрано, потому что управляемые атрибуты представляют собой количественные величины, которые в приложении должны быть положительны. В примере 22.27 показано, как выглядит класс `LineItem` с двумя свойствами, порожденными фабрикой `quantity`: для управления атрибутами `weight` и `price`.

Пример 22.27. bulkfood_v2prop.py: фабрика свойств `quantity` в действии

```
class LineItem:
    weight = quantity('weight') ❶
    price = quantity('price') ❷

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight ❸
        self.price = price

    def subtotal(self):
        return self.weight * self.price ❹
```

❶ Использовать фабрику для определения первого свойства, `weight`, в виде атрибута класса.
❷ Здесь создается второе свойство, `price`.
❸ Здесь свойство уже работает, и поэтому попытка присвоить `weight` нулевое или отрицательное значение отвергается.
❹ Здесь свойства также работают: с их помощью производится доступ к значениям, хранящимся в экземпляре.

Напомним, что свойства – атрибуты класса. При создании каждого свойства с помощью `quantity` мы должны передать имя атрибута `LineItem`, который будет управляться этим свойством. Необходимость дважды писать слово `weight` в следующей строке удручает:

```
weight = quantity('weight')
```

Но избежать такого повторения трудно, потому что свойство понятия не имеет, с каким атрибутом оно связывается. Помните: сначала вычисляется правая часть присваивания, поэтому в момент вызова функции `quantity()` атрибут класса `price` еще даже не существует.

Улучшить свойство `quantity`, так чтобы пользователю не приходилось дважды набирать имя атрибута, – нетривиальная задача метапрограммирования. Мы решим эту проблему в главе 23.

В примере 22.28 показана реализация фабрики свойств `quantity`[1].

Пример 22.28. bulkfood_v2prop.py: фабрика свойств `quantity`

```
def quantity(storage_name): ❶

    def qty_getter(instance): ❷
        return instance.__dict__[storage_name] ❸

    def qty_setter(instance, value): ❹
        if value > 0:
            instance.__dict__[storage_name] = value ❺
        else:
            raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter) ❻
```

❶ Аргумент `storage_name` определяет, где хранятся данные свойства; в случае свойства `weight` данные будут храниться в атрибуте с именем `'weight'`.

❷ Называть первый аргумент метода `qty_getter` именем `self` было бы не совсем правильно, т. к. это не тело класса; `instance` ссылается на экземпляр `LineItem`, в котором будет храниться атрибут.

❸ Метод `qty_getter` ссылается на `storage_name`, поэтому будет сохранен в замыкании этой функции; значение берется непосредственно из `instance.__dict__`, чтобы обойти свойство и избежать бесконечной рекурсии.

❹ В определении метода `qty_setter` первым аргументом также является `instance`.

❺ Значение сохраняется непосредственно в `instance.__dict__`, снова в обход свойства.

❻ Сконструировать и вернуть объект свойства.

Особого внимания заслуживают части этого кода, связанные с использованием переменной `storage_name`. Когда мы реализуем свойство традиционным способом, имя атрибута, в котором хранится значение, зашито в код методов чтения и установки. Здесь же функции `qty_getter` и `qty_setter` обобщенные, им необходима переменная `storage_name`, чтобы знать, из какого места атрибута `__dict__` экземпляра читать и в какое место записывать значение управляемого свойством атрибута. При каждом вызове фабрики `quantity` для порождения нового свойства переменная `storage_name` должна принимать уникальное значение.

[1] Идея кода заимствована из рецепта 9.21 «Как избежать повторения методов свойств» в книге David Beazley, Brian K. Jones «*Python Cookbook*», 3-е издание (O'Reilly).

Функции `qty_getter` и `qty_setter` обертываются объектом `property` в последней строке фабричной функции. Когда впоследствии любая из этих функций будет вызвана для выполнения своих обязанностей, она прочитает `storage_name` из своего замыкания и определит, откуда читать или куда записывать значение управляемого атрибута.

В примере 22.29, где я создаю и инспектирую экземпляр `LineItem`, видны атрибуты, в которых хранятся значения свойств.

Пример 22.29. bulkfood_v2prop.py: фабрика свойств `quantity`

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price ❶
(8, 13.95)
>>> nutmeg.__dict__ ❷
{'description': 'Moluccan nutmeg', 'weight': 8, 'price': 13.95
```

❶ Чтение `weight` и `price` с помощью свойств маскирует одноименные атрибуты экземпляра.

❷ Использовать метод `vars`, чтобы проинспектировать экземпляр `nutmeg`: видно, в каких точно атрибутах экземпляра хранятся значения.

Обратите внимание, как в свойствах, построенных нашей фабрикой, используется поведение, описанное в разделе «Свойства переопределяют атрибуты экземпляра» выше: свойство `weight` переопределяет атрибут экземпляра `weight`, поэтому любое обращение к `self.weight` или `nutmeg.weight` обрабатывается методами доступа свойства, и обойти их можно, только работая с атрибутом `__dict__` напрямую.

Возможно, код в примере 22.28 понятен не с первого раза, но он короткий: в нем столько же строк, сколько в паре декорированных методов чтения и установки одного лишь свойства `weight` в примере 22.21. Определение `LineItem` в примере 22.27 выглядит намного лучше без шума, вносимого методами чтения и установки.

В реальной системе такого рода проверкам могут подвергаться многие поля в нескольких классах, поэтому фабрику `quantity` следовало бы вынести в служебный модуль. В конечном итоге эту простую фабрику можно было бы заменить допускающим расширение дескрипторным классом, специализированные подклассы которого выполняют различные проверки. Мы займемся этим в главе 23.

А пока завершим обсуждение свойств, рассмотрев вопрос об удалении атрибутов.

Удаление атрибутов

Предложение `del` можно использовать для удаления не только переменных, но и атрибутов:

```
>>> class Demo:
...     pass
...
>>> d = Demo()
>>> d.color = 'green'
```

```
>>> d.color
'green'
>>> del d.color
>>> d.color
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Demo' object has no attribute 'color'
```

Удаление атрибутов вряд ли можно назвать повседневно выполняемой операцией, а требование обеспечить ее выполнение с помощью свойств еще более необычно. Но оно поддерживается, и я приведу для его демонстрации несколько искусственный пример.

В определении свойства декоратор `@my_propety.deleter` используется, чтобы обернуть метод, отвечающий за удаление атрибута, управляемого свойством. Идея глупого примера 22.30 подсказана персонажем Black Knight (Черный рыцарь) из скетча «Monty Python and the Holy Grail» (Монти Пайтон и Святой Грааль)[1].

Пример 22.30. blackknight.py

```
class BlackKnight:

    def __init__(self):
        self.phrases = [
            ('рука', "Это всего лишь царапина."),
            ('вторая рука', "Это всего лишь поверхностная рана."),
            ('нога', "Я неуязвим!"),
            ('вторая нога', "Ну ладно, пусть будет ничья.")
        ]

    @property
    def member(self):
        print('следующий член:')
        return self.phrases[0][0]

    @member.deleter
    def member(self):
        member, text = self.phrases.pop(0)
        print(f'ЧЕРНЫЙ РЫЦАРЬ (утрачена {member}) -- {text}')
```

Тесты для этого класса приведены в примере 22.31.

Пример 22.31. blackknight.py: тесты для примера 22.30 (Черный рыцарь никогда не признает поражение)

```
>>> knight = BlackKnight()
>>> knight.member
следующий член:
'рука'
>>> del knight.member
ЧЕРНЫЙ РЫЦАРЬ (утрачена рука) -- Это всего лишь царапина.
>>> del knight.member
ЧЕРНЫЙ РЫЦАРЬ (утрачена вторая рука) -- Это всего лишь поверхностная рана.
>>> del knight.member
ЧЕРНЫЙ РЫЦАРЬ (утрачена нога) -- Я неуязвим!
```

[1] По состоянию на октябрь 2021 года кровавая сцена доступа на YouTube (*https://www.youtube.com/watch?v=s35rVw1zskA*).

```
>>> del knight.member
ЧЕРНЫЙ РЫЦАРЬ (утрачена вторая нога) -- Ну ладно, пусть будет ничья.
```

Если используется не декоратор, а классический синтаксис, то для задания метода удаления применяется именованный аргумент `fdel`. Например, свойство `member` в теле класса `BlackKnight` можно было бы написать так:

```
member = property(member_getter, fdel=member_deleter)
```

Если вы не пользуетесь свойствами, то для удаления атрибута можно было бы также реализовать низкоуровневый специальный метод `__delattr__`, описанный в разделе «Специальные методы для управления атрибутами» ниже. Кодирование класса с применением метода `__delattr__` я оставляю в качестве упражнения на досуге для читателей.

Свойства – весьма полезный механизм, но иногда предпочтительнее более простые или низкоуровневые альтернативы. В последнем разделе этой главы мы рассмотрим некоторые базовые API, предлагаемые для программирования динамических атрибутов.

Важные атрибуты и функции для работы с атрибутами

И в этой главе, и раньше в книге мы уже использовали некоторые встроенные функции и специальные методы, которые Python предоставляет для работы с динамическими атрибутами. Сейчас мы соберем их в одном месте, поскольку в официальном руководстве они документированы в разных разделах.

Специальные атрибуты, влияющие на обработку атрибутов

Поведение многих функций и специальных методов, описанных ниже, определяется тремя специальными атрибутами.

`__class__`

Ссылка на класс объекта (т. е. `obj.__class__` – то же самое, что `type(obj)`). Python ищет специальные методы, например `__getattr__`, только в классе объекта, а не в самих экземплярах.

`__dict__`

Отображение, в котором хранятся изменяемые атрибуты объекта или класса. Если у объекта есть атрибут `__dict__`, то его в любой момент можно наделить новыми атрибутами. Если в классе есть атрибут `__slots__`, то у его экземпляров не может быть атрибута `__dict__`.

`__slots__`

Этот атрибут можно определить в классе, чтобы ограничить состав атрибутов у экземпляров этого класса. `__slots__` представляет собой кортеж строк с именами допустимых атрибутов[1]. Если имя `'__dict__'` отсутствует в `__slots__`, то у экземпляров класса не будет своего атрибута `__dict__`, по-

[1] Алекс Мартелли отмечает, что `__slots__` может быть и списком, но лучше не оставлять места для недоразумений и всегда использовать кортеж, потому что изменение списка, хранящегося в `__slots__`, после обработки тела класса интерпретатором, не возымеет никакого эффекта, так что использование здесь изменяемой последовательности лишь стало бы причиной вредных иллюзий.

этому в них будут разрешены только атрибуты, перечисленные в `__slots__`. См. дополнительные сведения в разделе «Экономия памяти с помощью атрибута класса __slots__» главы 11.

Встроенные функции для работы с атрибутами

Существует пять встроенных функций для чтения, записи и интроспекции атрибутов:

`dir([object])`

Перечисляет большую часть атрибутов объекта. В официальной документации (*https://docs.python.org/3/library/functions.html#dir*) сказано, что функция `dir` предназначена для интерактивного использования, поэтому она выводит не полный список атрибутов, а только самые «интересные». `dir` умеет инспектировать объекты с атрибутом `__dict__` и без него. Сам атрибут `__dict__` не входит в список, формируемый функцией `dir`, но ключи, хранящиеся в `__dict__`, входят. Есть еще несколько специальных атрибутов классов, в частности `__mro__`, `__bases__` и `__name__`, которые `dir` не выводит. Результат, печатаемый методом `dir`, можно модифицировать, реализовав специальный метод `__dir__`, как показано в примере 22.4. Если факультативный аргумент `object` не задан, то `dir` выводит имена в текущей области видимости.

`getattr(object, name[, default])`

Получает атрибут, идентифицируемый строкой `name`, объекта `object`. Применяется прежде всего для получения атрибутов (или методов), чьи имена заранее неизвестны. В результате может быть найден атрибут, определенный в классе или суперклассе объекта. Если такого атрибута не существует, `getattr` возбуждает исключение `AttributeError` либо возвращает значение `default`, если оно задано. Замечательный пример использования `getattr` имеется в методе `Cmd.onecmd` (*https://github.com/python/cpython/blob/19903085c3ad7a17c8047e1556c700f2eb109931/Lib/cmd.py#L214*) из стандартного пакета `cmd` – он служит для получения и выполнения заданной пользователем команды.

`hasattr(object, name)`

Возвращает `True`, если атрибут с указанным именем существует в объекте `object` или может быть найден с его помощью (например, в результате наследования). В документации (*https://docs.python.org/3/library/functions.html#hasattr*) приводится следующее объяснение: «Реализовано так: вызываем `getattr(object, name)`, а затем смотрим, возникло исключение `AttributeError` или нет».

`setattr(object, name, value)`

Присваивает значение `value` поименованному атрибуту `object`, если `object` это допускает. В результате может быть создан новый атрибут или изменен существующий.

`vars([object])`

Возвращает атрибут `__dict__` объекта `object`; функция `vars` не умеет работать с классами, в которых определен атрибут `__slots__` и нет атрибута `__dict__` (в отличие от функции `dir`, которая справляется с такими экземплярами). Без аргумента `vars()` делает то же самое, что `locals()`: возвращает словарь, описывающий локальную область видимости.

Специальные методы для работы с атрибутами

Специальные методы, описанные ниже, отвечают за чтение, установку, удаление и получение списка атрибутов (если они реализованы в пользовательском классе).

Доступ к атрибутам – с помощью нотации с точкой или встроенных функций `getattr`, `hasattr` и `setattr` – приводит к вызову соответствующих специальных методов. Чтение и запись атрибутов непосредственно в атрибуте `__dict__` экземпляра производятся в обход этих специальных методов – и это общепринятый метод обойти их в случае необходимости.

В разделе 3.3.11 «Поиск специальных методов» (*https://docs.python.org/3.10/reference/datamodel.html#special-method-lookup*) главы «Модель данных» есть такое предупреждение:

> Для пользовательских классов правильность работы при неявном вызове специальных методов гарантируется, только если они определены в типе объекта, а не в словаре экземпляра.

Иными словами, следует считать, что специальные методы ищутся в самом классе, даже если вызываются от имени экземпляра. По этой причине специальные методы не маскируются одноименными атрибутами экземпляра.

В следующих примерах предполагается, что существует класс с именем `Class`, что `obj` – экземпляр класса `Class`, а `attr` – атрибут `obj`.

Для всех описанных ниже специальных методов не имеет значения, как производится доступ к атрибуту: с помощью нотации с точкой или встроенных функций, упомянутых в предыдущем разделе. И `obj.attr`, и `getattr(obj, 'attr', 42)` приводят к вызову функции `Class.__getattribute__(obj, 'attr')`.

`__delattr__(self, name)`

Вызывается при любой попытке удалить атрибут в предложении `del`, например `del obj.attr` приводит к вызову `Class.__delattr__(obj, 'attr')`. Если `attr` является свойством, то его метод удаления никогда не вызывается, если в классе реализован метод `__delattr__`.

`__dir__(self)`

Вызывается при вызове `dir` для объекта с целью получить список атрибутов, например `dir(obj)` приводит к вызову `Class.__dir__(obj)`. Также используется для автоматического дополнения по нажатии клавиши **Tab** во всех современных консолях Python.

`__getattr__(self, name)`

Вызывается только тогда, когда попытка найти поименованный атрибут в `obj`, `Class` и суперклассах завершается неудачно. Выражения `obj.no_such_attr`, `getattr(obj, 'no_such_attr')` и `hasattr(obj, 'no_such_attr')` могут привести к вызову `Class.__getattr__(obj, 'no_such_attr')`, но только если атрибут с таким именем отсутствует в `obj`, `Class` и его суперклассах.

`__getattribute__(self, name)`

Вызывается при любой попытке получить поименованный атрибут непосредственно из Python-кода (интерпретатор иногда обходит этот метод, например чтобы получить метод `__repr__`). К вызову этого метода приводит использование нотации с точкой и встроенных функций `getattr` и `hasattr`.

Метод `__getattr__` всегда вызывается после `__getattribute__` и только в том случае, когда `__getattribute__` возбуждает исключение `AttributeError`. Чтобы при получении атрибутов `obj` не возникало бесконечной рекурсии, в реализации `__getattribute__` следует использовать `super().__getattribute__(obj, name)`.

`__setattr__(self, name, value)`

Вызывается при любой попытке установить поименованный атрибут. К вызову этого метода приводит использование нотации с точкой и встроенной функции `setattr`, например и `obj.attr = 42`, и `setattr(obj, 'attr', 42)` приводят к вызову `Class.__setattr__(obj, 'attr', 42)`.

Поскольку специальные методы `__getattribute__`и `__setattr__` вызываются безусловно и сопровождают практически каждый доступ к атрибуту, правильно использовать их труднее, чем метод `__getattr__`, который вызывается только для обработки имен несуществующих атрибутов. Во избежание ошибок лучше пользоваться не этими специальными методами, а свойствами или дескрипторами.

На этом завершается наше исследование свойств, специальных методов и других приемов программирования динамических атрибутов.

Резюме

Мы начали обсуждение динамических атрибутов с практических примеров классов, которые упрощают работу с набором данных в формате JSON. Первым примером был класс `FrozenJSON`, преобразующий вложенные словари и списки во вложенные экземпляры `FrozenJSON` и списки таких экземпляров. При этом мы продемонстрировали применение специального метода `__getattr__` для преобразования структур данных на лету, в момент чтения их атрибутов. В последней версии `FrozenJSON` было показано, как использовать метод конструирования `__new__`, чтобы превратить класс в гибкую фабрику объектов, причем не только этого класса.

Затем мы преобразовали набор JSON-данных в словарь `dict`, в которой хранятся сериализованные экземпляры класса `Record`. Первое воплощение `Record` содержало всего несколько строк, и в нем использовалась идиома `self.__dict__.update(**kwargs)` для создания произвольных атрибутов из именованных аргументов, переданных `__init__`. На второй итерации мы добавили класс `Event`, реализующий автоматический поиск связанных записей с помощью свойств. Вычисляемые свойства иногда необходимо кешировать, и мы рассмотрели несколько способов сделать это.

Знакомство со свойствами продолжилось на примере класса `LineItem`, в котором свойство предотвращало присваивание атрибуту `weight` нулевого или отрицательного значения. Глубже разобравшись с синтаксисом и семантикой свойств, мы создали фабрику свойств, которая обеспечивала одинаковую проверку свойств `weight` и `price`, но без повторного кодирования методов чтения и установки. При реализации фабрики свойств использовались тонкие идеи – замыкание и переопределение атрибутов экземпляра свойствами, – позволившие предложить элегантное общее решение, по количеству строк не превышающее определение одного свойства, написанное вручную.

Напоследок мы вкратце рассмотрели удаление атрибутов с помощью свойств, а затем перечислили специальные атрибуты, встроенные функции и специальные методы, которые поддерживают метапрограммирование атрибутов в Python.

Дополнительная литература

Официальной документацией по встроенным функциям для работы с атрибутами и интроспекции является глава 2 «Встроенные функции» (*https://docs.python.org/3/library/functions.html*) руководства по стандартной библиотеке Python. Относящиеся к этой же теме специальные методы и специальный атрибут `__slots__` документированы в разделе 3.3.2 «Настройка доступа к атрибутам» справочного руководства по языку Python (*https://docs. python.org/3/reference/datamodel.html#customizing-attribute-access*). Семантика вызова специальных методов в обход экземпляров описана в разделе 3.3.9 «Поиск специальных методов» (*https://docs.python.org/3/reference/datamodel.html#special-method-lookup*). В разделе 4.13 «Специальные атрибуты» главы 4 «Встроенные типы» руководства по стандартной библиотеке Python (*https://docs.python.org/3/library/stdtypes.html#special-attributes*) рассматриваются атрибуты `__class__` и `__dict__`.

В книге Дэвида Бизли и Брайана К. Джонса «*Python Cookbook*», 3-е издание (*https://www.oreilly.com/library/view/python-cookbook-3rd/9781449357337/*), есть несколько рецептов, относящихся к теме данной главы, но я упомяну только три наиболее интересных. Рецепт 8.8 «Расширение свойства в подклассе» касается непростого вопроса о переопределении методов внутри свойства, унаследованного от суперкласса. В рецепте 8.15 «Делегирование доступа к атрибутам» реализован прокси-класс, демонстрирующий большинство специальных методов, описанных в разделе «Специальные методы для работы с атрибутами» этой главы. А великолепный рецепт 9.21 «Как избежать повторения методов свойств» лег в основу фабрики свойств, представленной в примере 22.28.

В книге Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», 3-е издание (O'Reilly) (*https://www.oreilly.com/library/view/python-in-a/9781491913833/*), материал изложен строго и объективно. Авторы уделили свойствам всего три страницы, но это потому, что в книге принят аксиоматический стиль изложения: предшествующие 15 страниц посвящены детальному описанию семантики классов Python, начиная с самых основ, и в том числе дескрипторам, которые составляют основу реализации свойств. Так что, дойдя до свойств, авторы смогли уместить на трех страницах очень много полезной информации, в том числе и замечание, взятое мной в качестве эпиграфа к этой главе.

Бертран Мейер, чье определение *принципа единообразного доступа* приведено в начале этой главы, первым определил методологию проектирования по контракту, спроектировал язык Eiffel и написал великолепную книгу «Object-Oriented Software Construction», 2-е издание (Pearson). Первые шесть глав – одно из лучших концептуальных введений в объектно-ориентированный анализ и проектирование из всех мне встречавшихся. В главе 11 содержится введение в «проектирование по контракту», а в главе 35 – авторские оценки некоторых важнейших объектно-ориентированных языков: Simula, Smalltalk, CLOS (Common Lisp Object System), Objective-C, C++ и Java, а также краткие за-

мечания по поводу ряда других. Только на последней странице книги он признается, что «нотация», которой он пользовался при написании псевдокода, – на самом деле язык Eiffel.

Поговорим

Принцип единообразного доступа Мейера (любители акронимов иногда называют его UAP) эстетически весьма привлекателен. Как программисту, который пользуется некоторым API, мне должно быть все равно, что делает конструкция `coconut.price`: просто читает атрибут-данные или выполняет какое-то вычисление. Но как потребителю и гражданину мне это отнюдь небезразлично: в современной электронной коммерции значение `product.price` зачастую зависит от того, кто спрашивает, т. е. это заведомо не простой атрибут. На самом деле цена нередко будет ниже, если запрос поступает извне магазина, скажем от системы сравнения цен. И тем самым наказываются лояльные покупатели, которые любят гулять по данному магазину. Но это я отвлекся.

Однако это отвлечение поднимает важный для программирования вопрос: хотя в идеальном мире принцип единообразного доступа, безусловно, имеет смысл, на практике пользователям API иногда нужно знать, не может ли обращение к `product.price` оказаться слишком накладным или долгим. Это общая проблема абстракций программирования: они сильно затрудняют рассуждения о стоимости вычисления выражений во время выполнения. С другой стороны, абстракции позволяют пользователю сделать больше, написав меньше кода. Это компромисс. Как обычно, когда речь заходит о программной инженерии, стоит заглянуть на вики-сайт Уорда Каннингэма (*http://wiki.c2.com/?WelcomeVisitors*), где имеются поучительные соображения по поводу достоинств принципа единообразного доступа (*http://wiki.c2.com/?UniformAccessPrinciple*).

В объектно-ориентированных языках программирования применение или нарушение принципа единообразного доступа обычно сводится к выбору между чтением открытых атрибутов-данных и вызовом методов чтения и установки.

В Smalltalk и Ruby проблема решается просто и элегантно: они вообще не поддерживают открытые атрибуты-данные. Все атрибуты экземпляра в этих языках закрыты, поэтому доступ к ним обязательно опосредуется методами. Но это принуждение нивелируется удобным синтаксисом: в Ruby выражение `product.price` приводит к вызову метода чтения `price`; в Smalltalk мы пишем просто `product price`.

На другом конце спектра находится язык Java, в котором у программиста есть выбор между четырьмя модификаторами уровня доступа, причем подразумеваемый по умолчанию не именуется, а в пособии по Java (*https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html*) называется package-private.

Впрочем, принятая практика идет вразрез с синтаксисом, установленным проектировщиками языка Java. Все в мире Java согласны, что атрибуты должны быть закрытыми, поэтому приходится всякий раз писать `private`, поскольку этот уровень доступа не является умалчиваемым. Коль скоро все атрибуты закрыты, то и доступ к ним вне класса должен осуществляться с помощью акцессоров. В Java IDE имеются средства для автоматической генерации методов-акцессоров. К сожалению, IDE не поможет, когда спустя полгода вам придется читать свой код. Вам и только вам предстоит просеять кучу ничего не делающих акцессоров и найти те жемчужины, в которые имеет смысл добавить бизнес-логику.

Алекс Мартелли говорит от имени большей части сообщества Python, когда называет акцессоры «тупыми идиомами», и затем предлагает следующие примеры, которые выглядят совсем по-разному, но делают одно и то же[1]:

```
someInstance.widgetCounter += 1
# вместо...
someInstance.setWidgetCounter(someInstance.getWidgetCounter() + 1)
```

Иногда, проектируя API, я задавался вопросом, следует ли всякий метод, который не принимает аргументов (кроме `self`), возвращает значение (отличное от `None`) и является чистой функцией (т. е. не имеет побочных эффектов), заменять свойством, допускающим только чтение. В этой главе метод `LineItem.subtotal` (см. пример 19.23) был бы неплохим кандидатом на преобразование в свойство. Конечно, речь не идет о методах, которые призваны изменять объект, например `my_list.clear()`. Было бы ужасной ошибкой преобразовать такой метод в свойство, потому что простое обращение `my_list.clear` стерло бы все содержимое списка!

В библиотеке GPIO *Pingo* (*http://www.pingo.io/docs/*), одним из авторов которой я являюсь (упоминается в разделе «Метод __missing__» главы 3), значительная часть API пользовательского уровня основана на свойствах. Например, чтобы прочитать текущее значение аналогового контакта, нужно написать `pin.value`, а чтобы установить режим цифрового контакта – `pin.mode = OUT`. За кулисами то и другое может потребовать выполнения большого объема кода – в зависимости от драйвера платы. Мы решили использовать в Pingo свойства, потому что хотели, чтобы с API было удобно работать даже в интерактивных средах типа Jupyter Notebook, и полагали, что запись `pin.mode = OUT` приятнее и глазам, и пальцам, чем `pin.set_mode(OUT)`.

Решение, принятое в Smalltalk и Ruby, мне кажется чище, но я полагаю, что подход Python лучше, чем в Java. Нам разрешено начать с простого – сделать данные-члены открытыми атрибутами, – потому что мы знаем, что впоследствии всегда сможем обернуть их свойствами (или дескрипторами, о которых будем говорить в следующей главе).

`__new__` лучше, чем `new`

Еще один пример принципа единообразного доступа (или вариации на его тему) – тот факт, что в Python синтаксис вызова функций и создания объектов одинаков: `my_obj = foo()`, где `foo` может быть классом или любым другим вызываемым объектом.

В других языках, позаимствовавших синтаксис C++, имеется оператор `new`, из-за которого создание объекта выглядит иначе, чем вызов. По большей части пользователю API безразлично, является `foo` функцией или классом. На протяжении многих лет я и сам считал, что `property` – это функция. При обычном использовании это не важно.

Есть много причин заменить конструкторы фабриками[2]. Популярное обоснование – ограничить количество экземпляров, возвращая не новые, а соз-

[1] *Alex Martelli.* Python in a Nutshell. 2-е изд. O'Reilly. С. 101.

[2] Упоминаемые мной причины приведены в статье Джонатана Амстердама из журнала «Dr. Dobbs Journal» под названием «Java's new Considered Harmful» (*http://ubm.io/1cPP4PN*), а также в разделе «Consider static factory methods instead of constructors» удостоенной наград книги Джошуа Блоха «Effective Java», 3-е издание (Addison-Wesley).

данные ранее (как в паттерне Одиночка). Сюда же примыкает кеширование объектов, конструирование которых обходится дорого. Иногда также удобно возвращать объекты разных типов в зависимости от переданных аргументов.

Писать конструктор проще, но реализация фабрики повышает гибкость ценой дополнительного кода. В языках, где есть оператор `new`, проектировщик API должен заранее решить, ограничиться ли простым конструктором или потратить время на фабрику. Если первоначальное решение оказалось неверным, то исправление может обойтись дорого – из-за того, что `new` – оператор.

Иногда удобнее пойти другим путем и заменить простую функцию классом.

В Python классы и функции во многих ситуациях взаимозаменяемы. И не только из-за отсутствия оператора `new`, но и потому, что имеется специальный метод `__new__`, который позволяет преобразовать класс в фабрику, порождающую объекты разных видов (как мы видели в разделе «Гибкое создание объектов с помощью метода __new__» этой главы) или возвращающую ранее созданные экземпляры, вместо того чтобы каждый раз создавать новые.

Дуализм функции и класса было бы использовать еще проще, если бы в документе PEP 8 «Style Guide for Python Code» (*https://peps.python.org/pep-0008/#class-names*) не рекомендовалось применять `ВерблюжьюНотацию` (CamelCase) для имен классов. С другой стороны, в стандартной библиотеке имеются десятки классов с именами, составленными только из строчных букв (например, `property`, `str`, `defauldict` и т. д.). Так что, возможно, использование таких имен классов – вовсе не ошибка. Впрочем, как бы ни смотреть на эту проблему, разнобой в употреблении строчных и заглавных букв в именах классов из стандартной библиотеки Python представляет трудность для пользователей.

Хотя вызов функции не отличается от вызова класса, знать, что есть что, полезно, поскольку у классов есть дополнительная возможность: наследование. Поэтому лично я всегда применяю `ВерблюжьюНотацию` для имен своих классов и хотел бы, чтобы все классы в стандартной библиотеке следовали этому соглашению. Это я о вас, `collections.OrderedDict` и `collections.defaultdict`.

Глава 23

Дескрипторы атрибутов

> Изучение дескрипторов не только расширяет доступный инструментарий, но и позволяет глубже понять, как работает Python, и оценить элегантность его дизайна.
>
> – Раймонд Хэттингер, один из разработчиков ядра Python и гуру[1]

Дескрипторы – это способ повторного использования одной и той же логики доступа в нескольких атрибутах. Например, типы полей в объектно-ориентированных отображениях вроде Django ORM и SQL Alchemy – дескрипторы, управляющие потоком данных от полей в записи базы данных к атрибутам Python-объекта и обратно.

Дескриптор – это класс, который реализует динамический протокол, содержащий методы `__get__`, `__set__` и `__delete__`. Класс `property` реализует весь протокол дескриптора. Как обычно, разрешается реализовывать протокол частично. На самом деле большинство дескрипторов, встречающихся в реальных программах, реализуют только методы `__get__` и `__set__`, а многие – и вовсе лишь один из них.

Дескрипторы – уникальная черта Python, и используются они не только на уровне приложения, но и в инфраструктуре самого языка. Пользовательские функции – это дескрипторы. Мы увидим, как протокол дескрипторов делает методы связанными или несвязанными в зависимости от способа вызова.

Умение работать с дескрипторами – ключ к полному овладению Python. Им и посвящена эта глава.

В этой главе мы переработаем код примеров из раздела «Использование свойств для контроля атрибутов» главы 22, заменив свойства дескрипторами. Тем самым мы упростим повторное использование уже написанной ранее логики контроля атрибутов. Мы затронем понятия переопределяющих и непереопределяющих дескрипторов и поймем, что функции Python на самом деле являются дескрипторами. Наконец, мы немного расскажем о том, как реализуются дескрипторы.

Что нового в этой главе

Пример дескриптора `Quantity` в разделе «LineItem попытка № 4: автоматическая генерация имен атрибутов хранения» существенно упрощен, что стало возможным благодаря добавлению специального метода `__set_name__` в протокол дескрипторов в версии Python 3.6.

Я удалил пример фабрики свойств из раздела «LineItem попытка № 4: автоматическая генерация имен атрибутов хранения», поскольку он стал не ну-

[1] *Raymond Hettinger*. Descriptor HowTo Guide (*https://docs.python.org/3/howto/descriptor.html*).

жен: смысл его был в том, чтобы показать альтернативный способ решения проблемы `Quantity`, но после добавления метода `__set_name__` решение на основе дескриптора стало гораздо проще.

Класс `AutoStorage` из раздела «LineItem попытка № 5: новый тип дескриптора» тоже исключен, потому что утратил актуальность после добавления `__set_name__`.

Пример дескриптора: проверка значений атрибутов

В разделе «Программирование фабрики свойств» главы 19 мы видели, что фабрика свойств позволяет избежать многократного кодирования методов чтения и установки посредством применения приемов, характерных для функционального программирования. Фабрика свойств – это функция высшего порядка, которая создает параметризованный набор функций-акцессоров и строит из них экземпляры пользовательских свойств, настройки которых, например `storage_name`, хранятся в замыканиях. Объектно-ориентированный способ решения той же задачи – дескрипторный класс.

Мы вернемся к примерам класса `LineItem` с того места, где остановились, и переделаем фабрику свойств `quantity` в дескрипторный класс `Quantity`.

LineItem попытка № 3: простой дескриптор

Как было сказано во введении, класс, в котором реализован хотя бы один из методов `__get__`, `__set__` или `__delete__`, является дескриптором. Для использования дескриптора мы объявляем его экземпляры как атрибуты класса какого-то другого класса.

Мы создадим дескриптор `Quantity` и включим в класс `LineItem` два экземпляра `Quantity`: для управления атрибутами `weight` и `price`. Все это изображено на диаграмме классов на рис. 23.1.

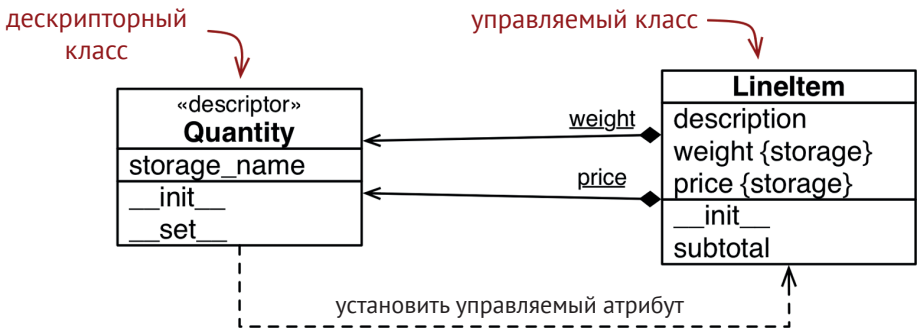


Рис. 23.1. UML-диаграмма класса `LineItem` и используемого в нем дескрипторного класса `Quantity`. Подчеркнуты атрибуты класса. Отметим, что `weight` и `price` – экземпляры класса `Quantity`, присоединенного к классу `LineItem`, но у экземпляров `LineItem` есть также собственные атрибуты `weight` и `price`, в которых соответствующие значения хранятся

Отметим, что слово `weight` встречается на рис. 23.1 дважды, потому что есть два разных атрибута с именем `weight`: первый – атрибут класса `LineItem`, второй – атрибут экземпляра, принадлежащий каждому объекту `LineItem`. То же самое относится и к `price`.

Начиная с этого места я буду пользоваться следующими определениями:

Дескрипторный класс

Класс, реализующий протокол дескриптора. Это класс `Quantity` на рис. 23.1.

Управляемый класс

Класс, в котором объявлены атрибуты класса, являющиеся экземплярами дескриптора. Это класс `LineItem` на рис. 23.1.

Экземпляр дескриптора

Любой экземпляр дескрипторного класса, объявленный атрибутом класса в управляемом классе. На рис. 23.1 все экземпляры дескриптора представлены стрелкой композиции, снабженной подчеркнутым именем (в UML подчеркивание означает атрибут класса). Сплошные ромбы одним концом касаются класса `LineItem`, который содержит экземпляры дескрипторов.

Управляемый экземпляр

Один экземпляр управляемого класса. В нашем примере управляемыми являются экземпляры класса `LineItem` (на диаграмме классов не показаны).

Атрибут хранения

Атрибут управляемого экземпляра, в котором хранится значение управляемого атрибута для данного экземпляра. На рис. 23.1 атрибутами хранения являются атрибуты `weight` и `price` экземпляра `LineItem`. Они отличаются от экземпляров дескриптора, которые всегда являются атрибутами класса.

Управляемый атрибут

Открытый атрибут управляемого класса, который обрабатывается экземпляром дескриптора, а значение которого хранится в одном из атрибутов хранения. Другими словами, экземпляр дескриптора и атрибут хранения в совокупности образуют инфраструктуру для управляемого атрибута.

Необходимо понимать, что экземпляры `Quantity` являются атрибутами класса `LineItem`. Этот важнейший момент иллюстрируется хреновинами и штуковинами на рис. 23.2.

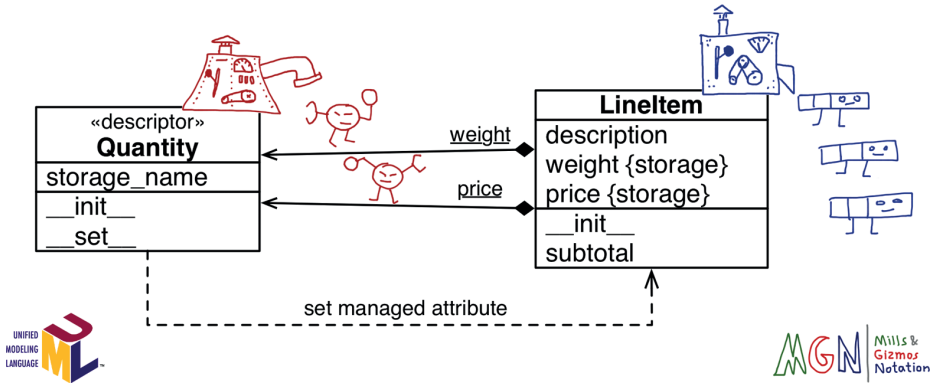


Рис. 23.2. UML-диаграмма классов, аннотированная на языке MGN (Mills & Gizmos Notation): классы представлены хреновинами, порождающими штуковины – экземпляры. Хреновина `Quantity` порождает две красные штуковины, присоединенные к хреновине `LineItem`: `weight` и `price`. Хреновина `LineItem` порождает синие штуковины, у которых есть собственные атрибуты `weight` и `price`, где хранятся значения

Введение в нотацию хреновин и штуковин

После многократного объяснения дескрипторов я понял, что UML – не лучший способ показа связей между классами и экземплярами, в частности связи между управляемым классом и экземплярами дескриптора[1]. Поэтому я изобрел собственный «язык», нотацию хреновин и штуковин (Mills & Gizmos Notation – MGN), который применяю для аннотирования UML-диаграмм.

Задача MGN – провести четкое различие между классами и экземплярами. Взгляните на рис. 23.3. В MGN класс изображается «хреновиной» (mill) – сложной машиной, которая производит «штуковины» (gizmo). Классы-хреновины всегда являются машинами с ручками и циферблатами. Штуковины – это экземпляры, они выглядят гораздо проще. Цвет штуковины всегда совпадает с цветом создавшей его хреновины (если книга напечатана в цвете).

Рис. 23.3. Набросок MGN, показывающий класс `LineItem` с тремя экземплярами и класс `Quantity` с двумя. Один экземпляр `Quantity` извлекает значение, хранящееся в экземпляре `LineItem`

Для рассматриваемого примера я изобразил экземпляр `LineItem` в виде строки табличного счета-фактуры с тремя колонками, представляющими три атрибута (`description`, `weight` и `price`). Поскольку экземпляры `Quantity` – дескрипторы, у них имеется лупа для получения значений методом `__get__` и клешня для установки значений методом `__set__`. Когда мы перейдем к метаклассам, вы еще скажете мне спасибо за эти каракули.

Но хватит рисовать каракули. Ниже приведен код: в примере 23.1 показан дескрипторный класс `Quantity`, а в примере 23.2 – новый класс `LineItem` с двумя экземплярами `Quantity`.

Пример 23.1. bulkfood_v3.py: дескриптор `Quantity` не принимает отрицательных значений

```
class Quantity: ❶

    def __init__(self, storage_name):
        self.storage_name = storage_name ❷
```

[1] Классы и экземпляры изображаются на UML-диаграммах классов прямоугольниками. Между ними есть визуальные различия, но экземпляры встречаются на диаграммах классов так редко, что разработчики их не отличают.

```
    def __set__(self, instance, value): ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value ❹
        else:
            msg = f'{self.storage_name} must be > 0'
            raise ValueError(msg)

    def __get__(self, instance, owner): ❺
        return instance.__dict__[self.storage_name]
```

❶ Дескриптор основан на протоколе, для его реализации не требуется наследование.

❷ В каждом экземпляре `Quantity` имеется атрибут `storage_name`: имя атрибута хранения, в котором хранится значение управляемого экземпляра.

❸ Метод `__set__` вызывается при любой попытке присвоить значение управляемому атрибуту. В данном случае `self` – экземпляр дескриптора (т. е. `LineItem.weight` или `LineItem.price`), `instance` – управляемый экземпляр (экземпляр `LineItem`), а `value` – присваиваемое значение.

❹ Мы должны сохранить значение атрибута непосредственно в `__dict__`; попытка вызвать `setattr(instance, self.storage_name)` привела бы к повторному вызову метода `__set__` и, стало быть, к бесконечной рекурсии.

❺ Реализовать `__get__` необходимо, потому что имя управляемого атрибута может не совпадать с `storage_name`. Про аргумент `owner` я расскажу ниже.

Реализация `__get__`требуется, потому что пользователь мог бы написать что-то вроде:

```
    class House:
        rooms = Quantity('number_of_rooms')
```

В классе `House` управляемым атрибутом является `rooms`, а атрибутом хранения `number_of_rooms`. Если имеется экземпляр `House` с именем `chaos_manor`, то чтение и запись `chaos_manor.rooms` проходят через дескриптор `Quantity`, присоединенный к `rooms`, но при чтении и записи `chaos_manor.number_of_rooms` дескриптор обходится.

Отметим, что `__get__` получает три аргумента: `self`, `instance` и `owner`. Аргумент `owner` – это ссылка на управляемый класс (например, `LineItem`), он полезен, если мы хотим, чтобы дескриптор поддерживал получение атрибута класса – быть может, чтобы эмулировать поведение Python по умолчанию – извлекать атрибут класса, если указанного имени нет среди атрибутов экземпляра.

Если управляемый атрибут, например `weight`, запрашивается через класс – `LineItem.weight`, – то метод `__get__` дескриптора получает `None` в качестве значения аргумента `instance`.

Для поддержки интроспекции и других приемов метапрограммирования пользователем рекомендуется возвращать из `__get__` экземпляр дескриптора, если доступ к управляемому атрибуту производится через класс. Для этого код `__get__` следовало бы написать так:

```
    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.storage_name]
```

В примере 23.2 демонстрируется использование класса `Quantity` в `LineItem`.

Пример 23.2. bulkfood_v3.py: дескрипторы `Quantity` управляют атрибутами в `LineItem`

```
class LineItem:
    weight = Quantity('weight') ❶
    price = Quantity('price') ❷

    def __init__(self, description, weight, price): ❸
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Первый экземпляр дескриптора связывается с атрибутом `weight`.
❷ Второй экземпляр дескриптора связывается с атрибутом `price`.
❸ Оставшаяся часть тела класса так же проста и понятна, как первоначальный код в файле *bulkfood_v1.py* (пример 22.19).

В примере 20.1 управляемые атрибуты называются так же, как соответствующий атрибут хранения, и никакой особой логики чтения нет, поэтому метод `__get__` в классе `Quantity` не нужен.

Код из примера 20.1 работает, как и ожидается, – не позволяет продать трюфели за 0 долларов[1]:

```
>>> truffle = LineItem('White truffle', 100, 0)
Traceback (most recent call last):
  ...
ValueError: value must be > 0
```

Кодируя метод `__set__`, не забывайте, что означают аргументы `self` и `instance`: `self` – это экземпляр дескриптора, а `instance` – управляемый экземпляр. Дескрипторы, управляющие атрибутами экземпляра, должны хранить значения в управляемых экземплярах. Потому-то Python и передает аргумент `instance` методам дескриптора.

Может возникнуть соблазн хранить значения всех управляемых атрибутов в экземпляре самого дескриптора, т. е. в методе `__set__` вместо кода

```
instance.__dict__[self.storage_name] = value
```

написать:

```
self.__dict__[self.storage_name] = value
```

Но это совершенно неправильно! Чтобы понять, почему, вспомните, что означают первые два аргумента `__set__`: `self` и `instance`. Здесь `self` – экземпляр

[1] Белые трюфели стоят тысячи долларов за фунт. Запрет продажи трюфелей за $0.01 оставляю в качестве упражнения для увлеченных читателей. Я знаю человека, который купил энциклопедию статистики, стоящую 1800 долларов, за 18 долларов из-за ошибки в ПО интернет-магазина (не Amazon.com).

дескриптора, т. е. фактически атрибут класса, принадлежащий управляемому классу. Одновременно в памяти могут находиться тысячи экземпляров `LineItem`, но экземпляров дескрипторов будет только два: `LineItem.weight` и `LineItem.price`. Поэтому все, что вы сохраняете в самих экземплярах дескрипторов, становится частью атрибута класса `LineItem` и, следовательно, распространяется на все экземпляры `LineItem`.

В примере 23.2 есть недостаток – необходимость повторять имена атрибутов, когда в теле управляемого класса создаются экземпляры дескрипторов. Хорошо было бы иметь возможность объявить класс `LineItem` как-то так:

```
class LineItem:
    weight = Quantity()
    price = Quantity()

    # прочие методы не изменяются
```

В версии из примера 23.2 задавать имя при каждом вызове `Quantity` необходимо явно, а это не только неудобно, но и опасно: если при копировании и вставке кода программист забудет изменить имена, т. е. напишет что-то вроде `price = Quantity('weight')`, то программа будет работать совершенно неправильно: затирать значение `weight` при изменении `price`.

Проблема в том – и мы видели это в главе 6, – что правая часть присваивания вычисляется еще до того, как начинает существовать переменная. Выражение `Quantity()` призвано создать экземпляр дескриптора, но в этот момент код в классе `Quantity` никак не может узнать имя переменной, с которой этот дескриптор должен быть связан (`weight` или `price`).

К счастью, протокол дескрипторов теперь поддерживает специальный метод, удачно названный `__set_name__`. Ниже мы увидим, как он используется.

Автоматическое генерирование имени атрибута хранения дескриптора раньше было неприятной проблемой. В первом издании этой книги я посвятил несколько страниц и строк кода в этой и следующей главах различным ее решениям, включая использование декоратора класса, а затем и метаклассов. В Python 3.6 все стало намного проще.

LineItem попытка № 4: автоматическое генерирование имен атрибутов хранения

Чтобы не набирать повторно имя атрибута в объявлении дескриптора, мы реализуем специальный метод `__set_name__`, который будет устанавливать атрибут `storage_name` в каждом экземпляре `Quantity`. Этот метод был добавлен в протокол дескрипторов в версии Python 3.6. Интерпретатор вызывает `__set_name__` для каждого дескриптора, который находит в теле `class`, – если дескриптор реализует его[1].

[1] Точнее, `__set_name__` вызывается из `type.__new__` – конструктора объектов, представляющих классы. Встроенный тип `type` на самом деле является метаклассом – классом по умолчанию для определенных пользователем классов. Поначалу это трудно переварить, но не беспокойтесь: глава 24 посвящена динамическому конфигурированию классов, в т. ч. и концепции метаклассов.

В примере 23.3 дескриптор `LineItem` не нуждается в методе `__init__`. Вместо этого `__set_name__` сохраняет имя атрибута хранения.

Пример 23.3. bulkfood_v4.py: `__set_name__` устанавливает имя для каждого экземпляра дескриптора `Quantity`

```
class Quantity:

    def __set_name__(self, owner, name): ❶
        self.storage_name = name ❷

    def __set__(self, instance, value): ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value
        else:
            msg = f'{self.storage_name} must be > 0'
            raise ValueError(msg)

  # __get__ не нужен ❹

class LineItem:
    weight = Quantity() ❺
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ `self` – экземпляр дескриптора (неуправляемый экземпляр), `owner` – управляемый класс, а `name` – имя атрибута `owner`, которому был назначен этот дескриптор в теле класса `owner`.

❷ Это то, что делал `__init__` в примере 23.1.

❸ Метод `__set__` здесь в точности такой же, как в примере 23.1.

❹ Реализовывать `__get__` необязательно, потому что имя атрибута хранения совпадает с именем управляемого атрибута. Выражение `product.price` получает атрибут `price` непосредственно из экземпляра `LineItem`.

❺ Теперь нам не нужно передавать имя управляемого атрибута конструктору `Quantity`. В этом и заключалась цель данной версии.

Глядя на пример 23.3, можно подумать, что кода слишком много для управления всего-то парой атрибутов, но важно понимать, что логика дескриптора теперь вынесена в отдельную кодовую единицу: класс `Quantity`. Обычно мы не определяем дескриптор в том же модуле, в каком он используется, а заводим отдельный служебный модуль, предназначенный для использования во всем приложении, а то и во многих приложениях, если разрабатывается библиотека или каркас.

С учетом этого пример 23.4 лучше демонстрирует типичное использование дескриптора.

Пример 23.4. bulkfood_v4c.py: ничем не загроможденное определение `LineItem`; дескрипторный класс `Quantity` теперь вынесен в импортированный модуль `model_v4c`

```
import model_v4c as model ❶

class LineItem:
    weight = model.Quantity() ❷
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Импортировать модуль `model_v4c`, в котором реализован дескриптор `Quantity`.
❷ Использовать `model.Quantity`.

Пользователи Django, наверное, заметили, что пример 23.4 выглядит в точности как определение модели. И это не случайность: поля моделей Django являются дескрипторами.

Поскольку дескрипторы определяются в классах, мы можем воспользоваться наследованием, чтобы повторно использовать уже написанный код в новых дескрипторах. Этим мы и займемся в следующем разделе.

LineItem попытка № 5: новый тип дескриптора

Воображаемый магазин натуральных пищевых продуктов столкнулся с неожиданной проблемой: каким-то образом была создана строка заказа с пустым описанием, и теперь заказ невозможно выполнить. Чтобы предотвратить такие инциденты в будущем, мы создадим новый дескриптор, `NonBlank`. Проектируя `NonBlank`, мы обнаруживаем, что он очень похож на дескриптор `Quantity`, а отличается только логика проверки.

Это наводит на мысль о рефакторинге и заведении двух базовых классов: завести абстрактный класс `Validated`, переопределяющий метод `__set__`, вызывая метод `validate`, который должен быть реализован в подклассах.

Затем мы переписываем `Quantity` и реализуем `NonBlank`, наследуя классу `Validated`, так что остается лишь написать методы `validate`.

Соотношение между классами `Validated`, `Quantity` и `NonBlank` – пример паттерна проектирования Шаблонный метод, который в классической книге «Паттерны проектирования» описывается следующим образом:

> Шаблонный метод определяет алгоритм в терминах абстрактных операций, которые переопределяются в подклассах для обеспечения конкретного поведения.

В примере 23.5 `Validated.__set__` – шаблонный метод, `self.validate` – абстрактная операция.

Пример 23.5. model_v5.py: абстрактный базовый класс `Validated`

```
import abc

class Validated(abc.ABC):

    def __set_name__(self, owner, name):
        self.storage_name = name

    def __set__(self, instance, value):
        value = self.validate(self.storage_name, value) ❶
        instance.__dict__[self.storage_name] = value ❷

    @abc.abstractmethod
    def validate(self, name, value): ❸
        """вернуть проверенное значение или возбудить ValueError"""
```

❶ Метод `__set__` делегирует проверку методу `validate` ...
❷ ... а затем использует возвращенное значение `value`, чтобы обновить хранимое значение.
❸ Метод `validate` абстрактный, это шаблонный метод.

Алекс Мартелли предпочитает называть этот паттерн проектирования *самоделегированием*, и я согласен, что это более подходящее название: первая строка `__set__` делегирует работу методу `validate` того же класса[1].

Конкретными подклассами `Validated` в этом примере являются `Quantity` и `NonBlank`, они показаны в примере 23.6.

Пример 23.6. model_v5.py: `Quantity` и `NonBlank` – конкретные подклассы `Validated`

```
class Quantity(Validated):
    """число, большее нуля"""

    def validate(self, name, value): ❶
        if value <= 0:
            raise ValueError(f'{name} must be > 0')
        return value

class NonBlank(Validated):
    """строка, содержащая хотя бы один символ, отличный от пробела"""

    def validate(self, name, value):
        value = value.strip()
        if not value: ❷
            raise ValueError(f'{name} cannot be blank')
      return value ❸
```

❶ Реализация шаблонного метода, которой требует абстрактный метод `Validated.validate`
❷ Если после удаления начальных и конечных пробелов ничего не осталось, отвергнуть значение.

[1] Слайд 50 лекции Алекса Мартелли «Python Design Patterns». Горячо рекомендую.

❸ Требуя, чтобы конкретные методы `validate` возвращали проверенное значение, мы оставляем им возможность очистить, преобразовать или нормализовать полученные данные. В данном случае значение `value` перед возвратом очищается от начальных и конечных пробелов.

Пользователям модуля *model_v5.py* все эти детали знать необязательно. Важно лишь, что они получают возможность использовать классы `Quantity` и `NonBlank` для автоматизации проверки атрибутов экземпляра. Последняя версия класса `LineItem` приведена в примере 20.7.

Пример 23.7. bulkfood_v5.py: использование дескрипторов `Quantity` и `NonBlank` в классе `LineItem`

```
import model_v5 as model ❶

class LineItem:
    description = model.NonBlank() ❷
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Импортировать модуль `model_v5` и попутно сопоставить ему более короткое имя.
❷ Использовать `model.NonBlank`. Больше ничего в коде не изменилось.

Приведенные в этой главе варианты класса `LineItem` демонстрируют типичное применение дескрипторов для управления атрибутами-данными. Дескриптор, подобный `Quantity`, называют еще переопределяющим, поскольку его метод `__set__` переопределяет (т. е. перехватывает и подменяет) установку одноименного атрибута в управляемом экземпляре. Однако существуют и непереопределяющие дескрипторы. Различие между ними мы подробно изучим в следующем разделе.

Переопределяющие и непереопределяющие дескрипторы

Напомним, что в способе обработки атрибутов в Python существует важная асимметрия. При чтении атрибута через экземпляр обычно возвращается атрибут, определенный в этом экземпляре, а если такого атрибута в экземпляре не существует, то атрибут класса. С другой стороны, в случае присваивания атрибуту экземпляра обычно создается атрибут в этом экземпляре, а класс вообще никак не затрагивается.

Эта асимметрия распространяется и на дескрипторы, в результате чего образуются две категории дескрипторов, различающиеся наличием или отсутствием метода `__set__`. Если `__set__` присутствует, то класс является переопределяющим дескриптором, иначе непереопределяющим. Эти термины начнут обретать смысл, когда мы будем изучать поведение дескрипторов в следующих примерах.

Чтобы увидеть отличия в поведении, нам понадобится несколько классов, и код в примере 23.8 станет нашим тестовым стендом.

Все методы `__get__` и `__set__` в примере 20.8 вызывают `print_args`, чтобы их вызовы были отчетливо видны. Понимать, как устроены вспомогательные функции `print_args`, `cls_name` и `display`, необязательно, так что не отвлекайтесь на них.

Пример 23.8. descriptorkinds.py: простые классы для изучения поведения переопределяющих и непереопределяющих дескрипторов

```
### вспомогательные функции для отображения ###

def cls_name(obj_or_cls):
    cls = type(obj_or_cls)
    if cls is type:
        cls = obj_or_cls
    return cls.__name__.split('.')[-1]

def display(obj):
    cls = type(obj)
    if cls is type:
        return f'<class {obj.__name__}>'
    elif cls in [type(None), int]:
        return repr(obj)
    else:
        return f'<{cls_name(obj)} object>'

def print_args(name, *args):
    pseudo_args = ', '.join(display(x) for x in args)
    print(f'-> {cls_name(args[0])}.__{name}__({pseudo_args})')

### существенные для этого примера классы ###

class Overriding: ❶
    """он же дескриптор данных или принудительный дескриптор"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner) ❷

    def __set__(self, instance, value):
        print_args('set', self, instance, value)

class OverridingNoGet: ❸
    """переопределяющий дескриптор без ``__get__``"""

    def __set__(self, instance, value):
        print_args('set', self, instance, value)

class NonOverriding: ❹
    """он же дескриптор без данных или маскируемый дескриптор"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)
```

```
class Managed: ❺
    over = Overriding()
    over_no_get = OverridingNoGet()
    non_over = NonOverriding()

    def spam(self): ❻
        print(f'-> Managed.spam({display(self)})')
```

❶ Типичный переопределяющий дескрипторный класс с методами `__get__` и `__set__`.
❷ В этом примере функция `print_args` вызывается из каждого метода дескриптора.
❸ Переопределяющий дескриптор без метода `__get__`.
❹ Здесь нет метода `__set__`, т. е. этот дескриптор непереопределяющий.
❺ Управляемый класс, в котором используется по одному экземпляру каждого дескрипторного класса.
❻ Метод `spam` включен для сравнения, потому что методы – также дескрипторы.

В следующих разделах мы исследуем поведение операций чтения и записи атрибутов класса `Managed` и одного его экземпляра с помощью каждого из определенных выше дескрипторов.

Переопределяющие дескрипторы

Дескриптор, в котором реализован метод `__set__`, называется *переопределяющим*, потому что, несмотря на то что этот дескриптор является атрибутом класса, он перехватывает все попытки присвоить значение атрибутам экземпляра. Именно так реализован дескриптор в примере 23.2. Свойства также являются переопределяющими дескрипторами: если мы не предоставим свою функцию установки, то по умолчанию будет использован метод `__set__` из класса `property`, который возбуждает исключение `AttributeError`, показывающее, что атрибут можно только читать. Эксперименты с переопределяющим дескриптором показаны в примере 23.9.

Разработчики Python и авторы книг и статей при обсуждении этих понятий используют разные термины. Я остановился на термине «переопределяющий дескриптор» из книги «Python in a Nutshell». В официальной документации употребляется термин «дескриптор данных», но «переопределяющий дескриптор» лучше отражает его специальное поведение. Переопределяющие дескрипторы также называют «принудительными дескрипторами». Синонимами термина «непереопределяющий дескриптор» являются «дескриптор без данных» и «маскируемый дескриптор».

Пример 23.9. Поведение переопределяющего дескриптора: `obj.over` – экземпляр класса `Overriding` (из примера 23.8)

```
>>> obj = Managed() ❶
>>> obj.over ❷
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
>>> Managed.over ❸
```

```
-> Overriding.__get__(<Overriding object>, None, <class Managed>)
>>> obj.over = 7 ❹
-> Overriding.__set__(<Overriding object>, <Managed object>, 7)
>>> obj.over ❺
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
>>> obj.__dict__['over'] = 8 ❻
>>> vars(obj) ❼
{'over': 8}
>>> obj.over ❽
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
```

❶ Создать объект `Managed` для тестирования.
❷ `obj.over` активирует метод дескриптора `__get__`, передавая ему управляемый экземпляр `obj` во втором аргументе.
❸ `Managed.over` активирует метод дескриптора `__get__`, передавая ему `None` во втором аргументе (`instance`).
❹ Присваивание `obj.over` активирует метод дескриптора `__set__`, передавая ему значение `7` в последнем аргументе.
❺ Чтение `obj.over` по-прежнему активирует метод дескриптора `__get__`.
❻ Установка значения непосредственно в `obj.__dict__` в обход дескриптора.
❼ Проверить, что значение попало в `obj.__dict__` и ассоциировано с ключом `over`.
❽ Однако даже при наличии атрибута экземпляра с именем `over` дескриптор `Managed.over` все равно переопределяет попытки читать `obj.over`.

Переопределяющий дескриптор без __get__

Свойства и другие переопределяющие дескрипторы, например поля моделей в Django, реализуют оба метода `__set__` и `__get__`, но, как мы видели в примере 23.2, можно также реализовать только `__set__`. В таком случае дескриптор обрабатывает только операцию записи. Чтение дескриптора через экземпляр вернет сам объект дескриптора, потому что не существует метода `__get__`, который мог бы перехватить эту операцию доступа. Если путем прямой записи в атрибут экземпляра `__dict__` был создан одноименный атрибут экземпляра с другим значением, то метод `__set__` все равно будет перехватывать последующие попытки изменить этот атрибут, однако его чтение просто вернет новое значение атрибута, а не объект дескриптора. Другими словами, атрибут экземпляра маскирует дескриптор, но только при чтении. См. пример 23.10.

Пример 23.10. Переопределяющий дескриптор без `__get__`

```
>>> obj.over_no_get ❶
<__main__.OverridingNoGet object at 0x665bcc>
>>> Managed.over_no_get ❷
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.over_no_get = 7 ❸
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get ❹
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.__dict__['over_no_get'] = 9 ❺
>>> obj.over_no_get
9
```

```
>>> obj.over_no_get = 7 ❼
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get ❽
9
```

❶ В этом переопределяющем дескрипторе нет метода `__get__`, поэтому чтение `obj.over_no_get` извлекает экземпляр дескриптора из класса.
❷ То же происходит, если извлечь экземпляр дескриптора непосредственно из управляемого класса.
❸ Попытка присвоить значение атрибуту `obj.over_no_get` активирует метод дескриптора `__set__`.
❹ Поскольку наш метод `__set__` не производит никаких изменений, повторное чтение `obj.over_no_get` извлекает все тот же экземпляр дескриптора из управляемого класса.
❺ Установить атрибут экземпляра с именем `over_no_get` через атрибут `__dict__` экземпляра.
❻ Теперь новый атрибут экземпляра `over_no_get` маскирует дескриптор, но только при чтении.
❼ Попытка присвоить значение атрибуту `obj.over_no_get` по-прежнему проходит через метод `__set__` дескриптора.
❽ Но при чтении дескриптор замаскирован до тех пор, пока существует одноименный атрибут экземпляра.

Непереопределяющий дескриптор

Дескриптор, в котором не реализован метод `__set__`, называется непереопределяющим. Установка атрибута экземпляра с таким же именем маскирует дескриптор, делая его бесполезным для обработки соответствующего атрибута в этом экземпляре. Методы реализованы как непереопределяющие дескрипторы. В примере 23.11 показана работа непереопределяющего дескриптора.

Пример 23.11. Поведение непереопределяющего дескриптора

```
>>> obj = Managed()
>>> obj.non_over ❶
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class Managed>)
>>> obj.non_over = 7 ❷
>>> obj.non_over ❸
7
>>> Managed.non_over ❹
-> NonOverriding.__get__(<NonOverriding object>, None, <class Managed>)
>>> del obj.non_over ❺
>>> obj.non_over ❻
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class Managed>)
```

❶ `obj.non_over` активирует метод дескриптора `__get__`, передавая ему `obj` во втором аргументе.
❷ `Managed.non_over` – непереопределяющий дескриптор, поэтому не существует метода `__set__`, который мог бы вмешаться в эту операцию присваивания.
❸ Теперь в `obj` есть атрибут экземпляра с именем `non_over`, который маскирует одноименный дескрипторный атрибут в классе `Managed`.

❹ Дескриптор `Managed.non_over` по-прежнему существует и перехватывает эту операцию доступа через класс.
❺ Если атрибут экземпляра `non_over` удалить…
❻ … то чтение `obj.non_over` активирует метод `__get__` дескриптора в классе, однако вторым аргументом будет управляемый экземпляр.

В предыдущих примерах мы видели несколько операций присваивания атрибуту экземпляра с таким же именем, как у дескриптора; результаты оказываются различны в зависимости от того, реализован в дескрипторе метод `__set__` или нет.

Установку атрибутов класса невозможно контролировать с помощью дескрипторов, присоединенных к тому же классу. В частности, это означает, что сами дескрипторные атрибуты можно затереть путем присваивания на уровне класса, как объясняется в следующем разделе.

Перезаписывание дескриптора в классе

Независимо от того, является дескриптор переопределяющим или нет, его можно перезаписать путем присваивания на уровне класса. Это техника партизанского латания, но в примере 23.12 дескрипторы подменяются целыми числами, что, безусловно, приведет к «поломке» любого класса, работа которого зависит от дескрипторов.

Пример 23.12. Дескриптор можно перезаписать в самом классе

```
>>> obj = Managed() ❶
>>> Managed.over = 1 ❷
>>> Managed.over_no_get = 2
>>> Managed.non_over = 3
>>> obj.over, obj.over_no_get, obj.non_over ❸
(1, 2, 3)
```

❶ Создать новый экземпляр для последующего тестирования.
❷ Перезаписать дескрипторные атрибуты в классе.
❸ Дескрипторов больше нет.

Пример 23.12 вскрывает еще одну асимметрию в чтении и записи атрибутов: хотя атрибут класса можно контролировать с помощью дескриптора с методом `__get__`, присоединенного к управляемому классу, но запись атрибута класса невозможно перехватить с помощью дескриптора с методом `__set__`, присоединенного к тому же классу.

Чтобы контролировать установку атрибутов класса, необходимо присоединить дескрипторы к классу класса – иначе говоря, к метаклассу. По умолчанию метаклассом всех пользовательских классов является `type`, а добавить атрибут в класс `type` невозможно. Но в главе 24 мы научимся создавать собственные метаклассы.

Теперь поговорим о том, как дескрипторы используются в Python для реализации методов.

Методы являются дескрипторами

Функция внутри класса становится связанным методом, потому что у всех определенных пользователем функций имеется метод `__get__`, а значит, будучи присоединены к классу, они ведут себя как дескрипторы. В примере 23.13 демонстрируется чтение метода `spam` из класса `Managed` примера 23.8.

Пример 23.13. Метод является непереопределяющим дескриптором

```
>>> obj = Managed()
>>> obj.spam ❶
<bound method Managed.spam of <descriptorkinds.Managed object at 0x74c80c>>
>>> Managed.spam ❷
<function Managed.spam at 0x734734>
>>> obj.spam = 7 ❸
>>> obj.spam
7
```

❶ Чтение `obj.spam` возвращает объект, представляющий связанный метод.
❷ Однако чтение `Managed.spam` возвращает функцию.
❸ Присваивание атрибуту `obj.spam` маскирует атрибут класса, делая метод `spam` недоступным через объект `obj`.

Поскольку в функциях не реализован метод `__set__`, они являются непереопределяющими дескрипторами, что видно из последней строки примера 23.13.

Еще отметим, что в примере 23.13 чтение `obj.spam` и `Managed.spam` дает разные объекты. Как для любых дескрипторов, метод `__get__` функции возвращает ссылку на себя, если доступ осуществляется через управляемый класс. Но при доступе через экземпляр метод `__get__` функции возвращает объект связанного метода: вызываемый объект, который обертывает функцию и связывает управляемый экземпляр (например, `obj`) с первым аргументом функции (т. е. `self`) – точно так же, как делает функция `functools.partial` (см. раздел «Фиксация аргументов с помощью functools.partial» главы 7). Чтобы лучше понять этот механизм, рассмотрим пример 23.14.

Пример 23.14. method_is_descriptor.py: класс `Text`, наследующий `UserString`

```
import collections

class Text(collections.UserString):

    def __repr__(self):
        return 'Text({!r})'.format(self.data)

    def reverse(self):
        return self[::-1]
```

Исследуем работу метода `Text.reverse`.

Пример 23.15. Эксперименты с методом

```
>>> word = Text('forward')
>>> word ❶
Text('forward')
```

```
>>> word.reverse() ❷
Text('drawrof')
>>> Text.reverse(Text('backward')) ❸
Text('drawkcab')
>>> type(Text.reverse), type(word.reverse) ❹
(<class 'function'>, <class 'method'>)
>>> list(map(Text.reverse, ['repaid', (10, 20, 30), Text('stressed')])) ❺
['diaper', (30, 20, 10), Text('desserts')]
>>> Text.reverse.__get__(word) ❻
<bound method Text.reverse of Text('forward')>
>>> Text.reverse.__get__(None, Text) ❼
<function Text.reverse at 0x101244e18>
>>> word.reverse ❽
<bound method Text.reverse of Text('forward')>
>>> word.reverse.__self__ ❾
Text('forward')
>>> word.reverse.__func__ is Text.reverse ❿
True
```

❶ Представление `repr` экземпляра `Text` выглядит как вызов конструктора `Text`, создающего точно такой же экземпляр.
❷ Метод `reverse` возвращает инвертированный текст.
❸ Метод, вызванный от имени класса, работает как функция.
❹ Обратите внимание на различие типов `function` и `method`.
❺ Метод `Text.reverse` работает как функция и применим даже к объектам, не являющимся экземплярами `Text`.
❻ Любая функция является непереопределяющим дескриптором. Если вызвать ее метод `__get__` от имени экземпляра, то будет возвращен метод, связанный с этим экземпляром.
❼ Если вызвать метод `__get__`, указав в качестве аргумента `instance` объект `None`, то будет возвращена сама функция.
❽ Выражение `word.reverse` приводит к вызову `Text.reverse.__get__(word)` и возврату связанного метода.
❾ У объекта связанного метода имеется атрибут `__self__`, в котором хранится ссылка на экземпляр, от имени которого вызывался метод.
❿ В атрибуте `__func__` связанного метода хранится ссылка на исходную функцию, присоединенную к управляемому классу.

У объекта связанного метода имеется метод `__call__`, который и отвечает за активацию. Этот метод вызывает исходную функцию, на которую ссылается атрибут `__func__`, передавая ей атрибут метода `__self__` в первом аргументе. Именно так работает неявное связывание с традиционным аргументом `self`.

Превращение функций в связанные методы – основной пример использования дескрипторов в инфраструктуре языка.

Разобравшись с тем, как работают дескрипторы и методы, дадим несколько практических советов по их использованию.

Советы по использованию дескрипторов

Ниже перечислены некоторые практические последствия только что описанных характеристик дескрипторов.

Для простоты пользуйтесь классом `property`

Встроенный класс `property` создает переопределяющие дескрипторы, в которых реализованы оба метода `__set__` и `__get__`, даже если вы сами не задавали метод установки[1]. Подразумеваемый по умолчанию метод `__set__` возбуждает исключение `AttributeError: can't set attribute`, поэтому свойство – это простейший способ создать доступный только для чтения атрибут и избежать проблемы, описанной ниже.

В дескрипторах только для чтения необходим метод `__set__`

Если вы используете дескрипторный класс для реализации атрибута, допускающего только чтение, то не забывайте реализовывать оба метода `__get__` и `__set__`, иначе одноименный атрибут экземпляра замаскирует дескриптор. Метод `__set__` атрибута, доступного только для чтения, должен просто возбуждать исключение `AttributeError` с подходящим сообщением[2].

Проверяющим дескрипторам достаточно одного метода `__set__`

Если дескриптор предназначен только для проверки значений, то метод `__set__` должен проверять полученный аргумент `value` и, если он правилен, устанавливать значение непосредственно в атрибуте `__dict__` экземпляра, используя в качестве ключа имя экземпляра дескриптора. Тогда чтение атрибута с таким же именем из экземпляра будет производиться максимально быстро, т. к. не требует наличия метода `__get__`. Код см. в примере 23.3.

Кеширование можно эффективно реализовать при наличии одного лишь `__get__`

Если вы напишете только метод `__get__`, то получите непереопределяющий дескриптор. Они полезны, когда требуется выполнить накладные вычисления и кешировать результат, установив атрибут экземпляра с таким же именем[3]. Одноименный атрибут экземпляра маскирует дескриптор, поэтому при последующем доступе к этому атрибуту значение будет извлекаться непосредственно из атрибута `__dict__` экземпляра в обход метода `__get__` дескриптора. Декоратор `@functools.cached_property` на самом деле порождает непереопределяющий дескриптор.

---

[1] Метод `__delete__` также предоставляется декоратором `property`, даже если вы сами не определяли никакого метода удаления.

[2] Python не блещет единообразием в таких сообщениях. При попытке изменить атрибут `c.real` комплексного числа выдается сообщение `AttributeError: read-only attribute`, а при попытке изменить `c.conjugate` (метод класса `complex`) – сообщение `AttributeError: 'complex' object attribute 'conjugate' is read-only`.

[3] Напомним, однако, что создание атрибутов экземпляра после выполнения метода `__init__` отключает оптимизацию разделения ключей (см. раздел «Практические последствия внутреннего устройства класса dict» главы 3).

Неспециальные методы можно замаскировать атрибутами экземпляра

Поскольку в функциях и методах реализован только метод `__get__`, они не перехватывают попытки установить одноименные атрибуты экземпляра, так что после простого присваивания `my_obj.the_method = 7` последующий доступ к `the_method` через данный экземпляр вернет число `7`, хотя на других экземплярах это никак не отразится. Однако на специальные методы это не распространяется. Интерпретатор ищет специальные методы только в самом классе, т. е. `repr(x)` всегда вычисляется как `x.__class__.__repr__(x)`, так что наличие атрибута `__repr__` в `x` не влияет на результат `repr(x)`. По той же причине существование атрибута с именем `__getattr__` в экземпляре не испортит обычный алгоритм доступа к атрибутам.

Может показаться, что простота переопределения неспециальных методов в экземплярах влечет за собой хрупкость дизайна и ошибки, но в моей более чем 20-летней практике программирования на Python это ни разу не приводило к проблемам. С другой стороны, если вы часто создаете динамические атрибуты, имена которых берутся из данных, не контролируемых вами (как в предыдущих главах этой книги), то об этом следует помнить и, наверное, включить какую-то фильтрацию или экранирование имен, чтобы динамические атрибуты имели смысл.

Класс `FrozenJSON` из примера 22.5 защищен от маскирования методов атрибутами экземпляра, потому что в нем есть только специальные методы и метод класса `build`. Методы класса безопасны, если обращение к ним производится только через класс, как в выражении `FrozenJSON.build` в примере 22.5 – которое я впоследствии заменил методом `__new__` в примере 22.6. Классы `Record` и `Event`, представленные в разделе «Вычисляемые свойства» главы 21, также безопасны: в них реализованы только специальные методы, статические методы и свойства. Свойства являются переопределяющими дескрипторами, поэтому не маскируются атрибутами экземпляра.

В заключение рассмотрим две особенности свойств, которые не были освещены в контексте дескрипторов: документирование и перехват попыток удалить управляемый атрибут.

Строка документации дескриптора и перехват удаления

Строка документации дескрипторного класса нужна для документирования экземпляров дескриптора в управляемом классе. На рис. 23.4 показано, как выглядит справка по классу `LineItem` с дескрипторами `Quantity` и `NonBlank` из примеров 23.6 и 23.7.

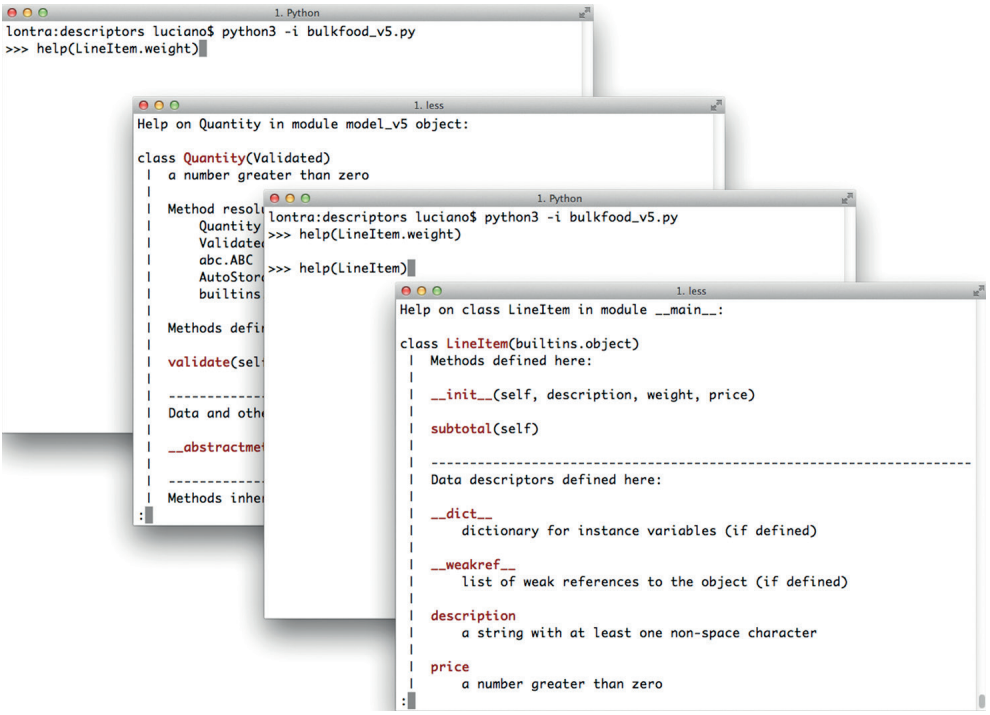


Рис. 23.4. Оболочка Python после выполнения команд `help(LineItem.weight)` и `help(LineItem)`

Это не вполне удовлетворительно. В случае `LineItem` неплохо было бы добавить информацию о том, что вес `weight` должен быть выражен в килограммах. Для свойств это тривиальная задача, потому что каждое свойство управляет одним конкретным атрибутом. Но дескрипторный класс `Quantity` используется для обоих атрибутов `weight` и `price`[1].

Вторая деталь, которую мы обсуждали для свойств, но опустили при рассмотрении дескрипторов, – перехват попыток удалить управляемый атрибут. Это можно сделать, реализовав метод `__delete__` вместе или вместо обычных методов `__get__` и (или) `__set__` в дескрипторном классе. Я сознательно опустил рассмотрение `__delete__`, потому что на практике он редко находит применение. Если понадобится, загляните в раздел «Реализация дескрипторов» документации по модели данных в Python (*https://docs.python.org/3.10/reference/datamodel.html# implementing-descriptors*). Написание «дурацкого» дескрипторного класса с методом `__delete__` оставляю в качестве упражнения досужему читателю.

[1] Задать текст справки для каждого экземпляра дескриптора на удивление трудно. Одно из возможных решений – динамически строить обертывающий класс для каждого экземпляра дескриптора.

Резюме

В начале главы мы продолжили тему класса `LineItem` из главы 22. В примере 23.2 мы заменили свойства дескрипторами. Мы видели, что дескриптор – это класс, экземпляры которого занимают место атрибутов в управляемом классе. Для обсуждения этого механизма пришлось ввести специальные термины, например *управляемый экземпляр* и *атрибут хранения*.

В разделе «LineItem попытка № 4: автоматическая генерация имен атрибутов хранения» мы отказались от требования явно задавать в объявлениях дескриптора `Quantity` параметр `storage_name`; это избыточно и чревато ошибками. Решение состоит в том, чтобы реализовать в классе `Quantity` специальный метод `__set_name__`, который будет сохранять имя управляемого свойства в виде `self.storage_name`.

В разделе «LineItem попытка № 5: новый тип дескриптора» мы показали, как унаследовать абстрактному дескрипторному классу для повторного использования кода при построении специализированных дескрипторов с пересекающейся функциональностью.

Затем мы изучили поведение дескрипторов, включающих и не включающих метод `__set__`, отметив принципиальное различие между переопределяющими и непереопределяющими дескрипторами. Проведя детальное тестирование, мы поняли, когда дескрипторы перехватывают управление, а когда маскируются, обходятся или затираются.

После этого исследовали специальную категорию непереопределяющих дескрипторов: методы. Тестирование на консоли показало, как благодаря протоколу дескрипторов присоединенная к классу функция становится методом при обращении через экземпляр.

Наконец, в разделе «Советы по использованию дескрипторов» было представлено несколько практических советов, а в разделе «Строка документации дескриптора и перехват удаления» приведены краткие сведения о документировании дескрипторов.

В разделе «Что нового в этой главе» было отмечено, что несколько примеров стали значительно проще благодаря специальному методу `__set_name__` протокола дескрипторов, который был добавлен в версии Python 3.6. Так работает эволюция языка!

Дополнительная литература

Помимо официального справочного материала в главе «Модель данных» (*https://docs.python.org/3/reference/datamodel.html*), ценным ресурсом является пособие Раймонда Хэттингера «Descriptor HowTo Guide» (*https://docs.python.org/3/howto/descriptor.html*), оно входит в подборку практических руководств (*https://docs.python.org/3/howto/descriptor.html*), являющуюся частью официальной документации по Python.

Как и во всем, что касается объектной модели в Python, книга Martelli, Ravenscroft, Holden «Python in a Nutshell», 3-е издание (O'Reilly), является авторитетным и объективным источником. Мартелли также подготовил презен-

тацию «Python's Object Model», в которой всесторонне рассматриваются свойства и дескрипторы (слайды – по адресу *http://www.aleax.it/Python/nylug05_om.pdf*, видео – по адресу *https://www.youtube.com/watch?v=VOzvpHoYQoo*).

Имейте в виду, что любое описание дескрипторов, созданное до того, как в 2016 году был одобрен документ PEP 487, скорее всего, содержит примеры, которые теперь можно считать чрезмерно усложненными, потому что до версии 3.6 метод `__set_name__` не поддерживался.

За дополнительными практическими примерами обратитесь к книге Дэвида Бизли и Брайана К. Джонса «*Python Cookbook*», 3-е издание (O'Reilly), где есть много рецептов, иллюстрирующих дескрипторы. Особо мне хочется отметить рецепты 6.12 «Чтение вложенных структур и имеющих переменную длину двоичных структур», 8.10 «Свойства с отложенным вычислением», 8.13 «Реализация модели данных или системы типов» и 9.9 «Определение декораторов как классов». В последнем рецепте глубоко освещаются вопросы взаимодействия между декораторами функций, дескрипторами и методами и объясняется, почему декоратор функции, реализованный в виде класса с методом `__call__`, должен также реализовывать метод `__get__`, если его предполагается применять для декорирования не только функций, но и методов.

В документе PEP 487 «Simpler customization of class creation» (*https://peps.python.org/pep-0487/*) описан специальный метод `__set_name__` и имеется пример проверяющего дескриптора.

Поговорим

Дизайн self

Требование явно объявлять `self` первым аргументом методов – одно из противоречивых проектных решений в Python. После 23 лет использования языка я к нему привык. Я думаю, что это решение – пример философии проектирования «чем хуже, тем лучше», описанной Ричардом П. Гэбриелом в работе «The Rise of Worse is Better» (*https://dreamsongs.com/RiseOfWorseIsBetter.html*). Важнейший приоритет в этой философии – «простота». Вот как это звучит в изложении Гэбриела:

> Реализация и интерфейс должны быть простыми. Простота реализации даже важнее простоты интерфейса. Простота – самое важное требование при выборе дизайна.

Явный `self` в Python – воплощение этой философии проектирования. Простота – даже элегантность – реализации достигается за счет пользовательского интерфейса: сигнатура метода – `def zfill(self, width)` – визуально не соответствует его вызову – `label.zfill(8)`.

Это соглашение – и использование идентификатора `self` – впервые появилось в языке Modula-3, но есть и отличие: в Modula-3 интерфейсы объявляются отдельно от реализации, и в объявлении интерфейса аргумент `self` опущен, поэтому, с точки зрения пользователя, у метода в объявлении интерфейса ровно столько же аргументов, сколько задается при его вызове.

Со временем сообщения об ошибках, касающиеся аргументов методов, стали понятнее. Если вызывается определенный пользователем метод с одним аргументом, кроме `self`, – `obj.meth()`, – то в Python 2.7 возбуждается исключение

```
TypeError: meth() takes exactly 2 arguments (1 given)
 (meth() принимает ровно 2 аргумента (задан 1)),
```

тогда как в Python 3 непонятно откуда взявшийся аргумент не упоминается, а указывается имя недостающего аргумента:

```
TypeError: meth() missing 1 required positional argument: 'x'
(при вызове meth() не задан 1 обязательный позиционный аргумент: ‘x’).
```

Помимо использования `self` в качестве явного аргумента, мишенью для критики часто становится требование указывать его при любом доступе к атрибутам экземпляра. См., например, знаменитое сообщение А. М. Кухлинга «Бородавки Python» (*http://web.archive.org/web/20031002184114/www.amk.ca/python/writing/warts.html*); самого Кухлинга не очень беспокоит квалификатор `self`, но он упоминает эту проблему в числе прочих, быть может, отражая мнения, высказанные в группе `comp.lang.python`. Лично меня необходимость набирать `self` не раздражает: я считаю, что полезно отличать локальные переменные от атрибутов. Я больше возражаю против использования `self` в предложении `def`.

Всякий, кому не нравится явное использование `self` в Python, отнесется к нему гораздо снисходительнее, если взглянет на путаную семантику (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this*) неявного `this` в JavaScript. У Гвидо были основательные причины спроектировать `self` именно таким образом, и он написал о них в своем блоге «История Python» в статье «Adding Support for User-Defined Classes» (*http://python-history.blogspot.com/2009/02/adding-support-for-user-defined-classes.html*).

Глава 24

Метапрограммирование классов

> Известно, что отладка в два раза сложнее написания программы. Поэтому если вы были предельно хитроумны при написании программы, то что же вы будете делать при ее отладке?
>
> Брайан У. Керниган, Ф. Дж. Плоджер, «Элементы стиля программирования»[1]

Метапрограммирование классов – это искусство создания или настройки классов во время выполнения. Классы в Python – полноправные объекты, поэтому функция может в любой момент создать новый класс, не используя ключевое слово `class`. Декораторы классов – также функции, которые дополнительно умеют инспектировать и изменять декорированный класс и даже заменять его другим. Наконец, метаклассы – самое продвинутое средство метапрограммирования классов: они позволяют создавать целые категории классов со специальными характеристиками, например уже встречавшиеся нам абстрактные базовые классы.

Метаклассы – мощный механизм, но оправдать его применение трудно, а правильно пользоваться еще труднее. Декораторы классов решают многие из тех же проблем, а понять их проще. Кроме того, в Python 3.6 реализован документ PEP 487 «Simpler customization of class creation» (*https://peps.python.org/pep-0487/*), в котором описаны специальные методы, поддерживающие задачи, которые раньше требовали метаклассов или декораторов[2].

В этой главе приемы метапрограммирования изложены в порядке возрастания сложности.

Эта тема настолько завораживает, что легко увлечься. Поэтому в самом начале главы я обязан дать следующий совет.

Ради удобства чтения и сопровождения программы лучше откажитесь от использования описанных в этой главе приемов в коде приложения.

С другой стороны, это именно то, что надо, если вы собираетесь написать очередной великий каркас на Python.

---

[1] Цитата из главы 2 книги «Элементы стиля программирования», 2-е издание (М.: Радио и связь, 1984, пер. с англ. В. А. Волынского).

[2] Это не значит, что из-за PEP 487 код, в котором эти средства использовались, перестал работать. Просто код, в котором до версии Python 3.6 использовались метаклассы или декораторы классов, теперь можно переработать, сделав проще и, возможно, эффективнее.

Что нового в этой главе

Весь код из главы «Метапрограммирование классов» в первом издании книги по-прежнему работает правильно. Но некоторые примеры уже нельзя назвать самыми простыми решениями в свете новых возможностей, добавленных в версии Python 3.6 и последующих.

Я заменил эти примеры другими, уделив внимание новым средствам метапрограммирования в Python или добавив новые требования, чтобы оправдать использование более продвинутых приемов. В некоторых новых примерах используются аннотации типов, чтобы предоставить построители классов, аналогичные декоратору `@dataclass` и типу `typing.NamedTuple`.

Раздел «Метаклассы на практике» новый, в нем рассматриваются некоторые общие соображения о применимости метаклассов.

Один из лучших способов рефакторинга – удалить код, ставший излишним вследствие появления новых, более простых способов решения той же задачи. Это относится в равной мере к коду и к книгам.

Мы начнем с обзора атрибутов и методов, определенных в модели данных Python для всех классов.

Классы как объекты

Как и большинство программных сущностей в Python, классы являются объектами. Каждый класс имеет ряд атрибутов, определенных в модели данных Python и документированных в разделе 4.13 «Специальные атрибуты» (*https://docs.python.org/3/library/stdtypes.html#special-attributes*) главы «Встроенные типы» справочного руководства по стандартной библиотеке. Три из них мы уже встречали ранее: `__class__`, `__name__` и `__mro__`. Перечислим остальные.

`cls.__bases__`

Кортеж, содержащий базовые классы данного класса.

`cls.__qualname__`

Полное имя класса или функции, представляющее собой путь от глобальной области видимости модуля к определению класса, компоненты которого разделены точками. Так, в классе из модели Django, например `Ox` (*https://docs.djangoproject.com/en/3.2/topics/db/models/#meta-options*), имеется внутренний класс с именем `Meta`. Атрибут `__qualname__` класса `Meta` равен `Ox.Meta`, тогда как `__name__` равен просто `Meta`. Спецификация этого атрибута приведена в документе PEP-3155 «Qualified name for classes and functions» (*http://www.python.org/dev/peps/pep-3155*).

`cls.__subclasses__()`

Этот метод возвращает список непосредственных подклассов данного класса. В реализации применяются слабые ссылки, чтобы избежать циклических ссылок между суперклассом и его подклассами, – которые хранят сильную ссылку на суперклассы в атрибуте `__bases__`. В возвращенный спи-

сок включаются только подклассы, которые в настоящий момент загружены в память. Подклассы в еще не загруженных модулях не видны.

`cls.mro()`

Интерпретатор вызывает этот метод при построении класса, чтобы получить кортеж суперклассов, который хранится в атрибуте класса `__mro__`. Метакласс может переопределить этот метод и задать другой порядок разрешения методов в конструируемом классе.

Ни один из упоминаемых в этом разделе атрибутов не включается в список, возвращаемый функцией `dir(…)`.

Но если класс – это объект, то что является классом класса?

TYPE: ВСТРОЕННАЯ ФАБРИКА КЛАССОВ

Обычно мы воспринимаем `type` как функцию, которая возвращает класс объекта, потому что именно это делает выражение `type(my_object)`: возвращает `my_object.__class__`.

Однако `type` – это класс, который создает новый класс, если вызывается с тремя аргументами.

Рассмотрим следующий простой класс:

```
class MyClass(MySuperClass, MyMixin):
    x = 42

    def x2(self):
        return self.x * 2
```

С помощью конструктора `type` мы можем создать `MyClass` во время выполнения:

```
MyClass = type('MyClass',
               (MySuperClass, MyMixin),
               {'x': 42, 'x2': lambda self: self.x * 2},
          )
```

Этот вызов `type` функционально эквивалентен предыдущему предложению блока `class MyClass…`.

Прочитав предложение `class`, Python вызывает `type`, чтобы построить объект класса, и передает ему следующие параметры:

`name`

Идентификатор, расположенный после ключевого слова `class`, например `MyClass`.

`bases`

Кортеж суперклассов, расположенный в скобках после идентификатора класса, или `(object,)`, если в предложении `class` нет суперклассов.

`dict`

Отображение имен атрибутов на значения. Вызываемые объекты становятся методами, как мы видели в разделе «Методы являются дескрипторами» главы 23. Другие значения становятся атрибутами класса.

Конструктор `type` принимает факультативные именованные аргументы, которые игнорируются самим `type`, но передаются без изменения методу `__init_subclass__`, который должен их потребить. Этот специальный метод мы изучим в разделе «Введение в __init_subclass__» ниже, но об использовании именованных аргументов я рассказывать не буду. Дополнительные сведения см. в документе PEP 487 «Simpler customization of class creation» (*https://peps.python.org/pep-0487/*).

Классом `type` является *метакласс*: класс, который строит классы. Иными словами, экземплярами класса `type` являются классы. В стандартной библиотеке есть и другие метаклассы, но `type` подразумевается по умолчанию:

```
>>> type(7)
<class 'int'>
>>> type(int)
<class 'type'>
>>> type(OSError)
<class 'type'>
>>> class Whatever:
... pass
...
>>> type(Whatever)
<class 'type'>
```

Мы будем строить собственные метаклассы в разделе «Основы метаклассов» ниже.

А следующей нашей темой будет использование встроенного типа `type` для создания функции, которая строит классы.

Функция-фабрика классов

В стандартной библиотеке есть фабрика классов, с которой мы уже неоднократно встречались: `collections.namedtuple`. В главе 5 мы также видели `typing.NamedTuple` и `@dataclass`. Во всех этих построителях классов используется техника, рассматриваемая в этой главе.

Мы начнем с простой фабрики классов изменяемых объектов – простейшей возможной замены декоратора `@dataclass`.

Предположим, что мы пишем приложение для зоомагазина и хотим обрабатывать данные о собаках как простые записи. Плохо, если придется писать такой стереотипный код:

```
class Dog:
    def __init__(self, name, weight, owner):
        self.name = name
        self.weight = weight
        self.owner = owner
```

Нудно-то как... каждое имя поля встречается по три раза. И даже симпатичного представления `repr` мы при этом не получили:

```
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex
<__main__.Dog object at 0x2865bac>
```

Позаимствовав идею у `collections.namedtuple`, напишем функцию `record_factory`, которая будет создавать простые классы вроде `Dog` на лету. В примере 24.1 показано, как она должна работать.

Пример 24.1. Тестирование `record_factory`, простой фабрики классов

```
>>> Dog = record_factory('Dog', 'name weight owner') ❶
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex ❷
Dog(name='Rex', weight=30, owner='Bob')
>>> name, weight, _ = rex ❸
>>> name, weight
('Rex', 30)
>>> "{2}'s dog weighs {1}kg".format(*rex) ❹
"Bob's dog weighs 30kg"
>>> rex.weight = 32 ❺
>>> rex
Dog(name='Rex', weight=32, owner='Bob')
>>> Dog.__mro__ ❻
(<class 'factories.Dog'>, <class 'object'>)
```

❶ Фабрику можно вызывать как `namedtuple`: имя класса, а за ним строка имен атрибутов через пробел.
❷ Удобное представление `repr`.
❸ Экземпляры являются итерируемыми объектами, поэтому их можно распаковывать в момент присваивания ...
❹ ... или при передаче функциям типа `format`.
❺ Экземпляр записи изменяемый.
❻ Вновь созданный класс наследует `object` – никакой связи с нашей фабрикой нет.

Код `record_factory` приведен в примере 24.2[1].

Пример 24.2. record_factory.py: простая фабрика классов

```
from typing import Union, Any
from collections.abc import Iterable, Iterator

FieldNames = Union[str, Iterable[str]] ❶

def record_factory(cls_name: str, field_names: FieldNames) -> type[tuple]: ❷

    slots = parse_identifiers(field_names) ❸

    def __init__(self, *args, **kwargs) -> None: ❹
        attrs = dict(zip(self.__slots__, args))
        attrs.update(kwargs)
```

[1] Спасибо моему другу Х. С. Буэно, предложившему это решение.

```
        for name, value in attrs.items():
            setattr(self, name, value)

    def __iter__(self) -> Iterator[Any]: ❺
        for name in self.__slots__:
            yield getattr(self, name)

    def __repr__(self): ❻
        values = ', '.join(f'{name}={value!r}'
            for name, value in zip(self.__slots__, self))
        cls_name = self.__class__.__name__
        return f'{cls_name}({values})'

    cls_attrs = dict( ❼
        __slots__=slots,
        __init__=__init__,
        __iter__=__iter__,
        __repr__=__repr__,
    )

    return type(cls_name, (object,), cls_attrs) ❽

def parse_identifiers(names: FieldNames) -> tuple[str, ...]:
    if isinstance(names, str):
        names = names.replace(',', ' ').split() ❾
    if not all(s.isidentifier() for s in names):
        raise ValueError('names must all be valid identifiers')
    return tuple(names)
```

❶ Пользователь может предоставить имена полей в виде одной строки или итерируемого объекта строк.
❷ Принимаем такие же аргументы, как первые два в `collections.namedtuple`; возвращаем `type`, т. е. класс, который ведет себя как кортеж.
❸ Построить кортеж имен атрибутов, он станет атрибутом `__slots__` нового класса.
❹ Эта функция станет методом `__init__` в новом классе. Она принимает позиционные и (или) именованные аргументы[1].
❺ Отдавать значения полей в порядке, определяемом атрибутом `__slots__`.
❻ Породить удобное представление, обходя `__slots__` и `self`.
❼ Построить словарь атрибутов класса.
❽ Построить и вернуть новый класс, вызывая конструктор `type`.
❾ Преобразовать строку `names`, в которой имена разделены пробелами или запятыми, в список строк.

Пример 24.2 – первый случай, когда мы встретили `type` в аннотации типа. Если бы аннотация содержала просто `-> type`, это значило бы, что `record_factory` возвращает класс, – и было бы правильно. Но аннотация `-> type[tuple]` более точна: она говорит, что возвращенный класс будет подклассом `tuple`.

[1] Я не стал добавлять аннотации типов для аргументов, потому что фактические типы – `Any`. Аннотацию типа возвращаемого значения я включил, потому что иначе Муру не стала бы заглядывать внутрь метода.

В последней строке функции `record_factory` строится класс с именем, равным значению `cls_name`. Он наследует `object`, а в его пространство имен загружены атрибуты `__slots__`, `__init__`, `__iter__` и `__repr__`, последние три из которых являются методами экземпляра.

Мы могли бы назвать атрибут класса `__slots__` как-то иначе, но тогда пришлось бы реализовывать метод `__setattr__`, проверяющий имена атрибутов, которым присваиваются значения, потому что мы хотим, чтобы в наших классах, подобных записям, набор атрибутов всегда был один и тот же и чтобы атрибуты в нем следовали в одном и том же порядке. Напомню, однако, что основное назначение `__slots__` – экономия памяти в случае, когда экземпляров миллионы, и что использование `__slots__` сопряжено с некоторыми недостатками, описанными в разделе «Экономия памяти с помощью атрибута класса __slots__» главы 11.

Экземпляры классов, созданных с помощью `record_factory`, не сериализуемы, т. е. их нельзя экспортировать функцией `dump` из модуля `pickle`. Решение этой проблемы выходит за рамки рассматриваемого примера, цель которого – продемонстрировать использование класса `type` в простом случае. Полное решение смотрите в исходном коде `collections.namedtuple` (*https://github.com/python/cpython/blob/3.9/Lib/collections/__init__.py*); ищите слово «pickling».

Теперь посмотрим, как эмулировать более современные построители классов, например `typing.NamedTuple`, который принимает пользовательский класс в виде предложения `class` и автоматически наделяет его дополнительной функциональностью.

Введение в __init_subclass__

Оба метода, `__init_subclass__` и `__set_name__`, были предложены в документе PEP 487 «Simpler customization of class creation» (*https://peps.python.org/pep-0487/*). С применением специального метода `__set_name__` в дескрипторах мы познакомились в разделе «LineItem попытка № 4: автоматическое генерирование имен атрибутов хранения» главы 23. А теперь рассмотрим метод `__init_subclass__`.

В главе 5 мы видели, что `typing.NamedTuple` и `@dataclass` позволяют использовать предложение `class` для задания атрибутов нового класса, в который построитель классов затем добавляет методы `__init__`, `__repr__`, `__eq__` и т. д.

Чтобы дополнить класс, оба построителя классов читают аннотации типов в заданном пользователем предложении `class`. Те же самые аннотации позволяют программам проверки типов проверить код, который читает или устанавливает эти атрибуты. Однако `NamedTuple` и `@dataclass` не пользуются аннотациями типов для проверки атрибутов во время выполнения. Зато это делает класс `Checked`, рассматриваемый в следующем примере.

Невозможно поддержать любую мыслимую статическую аннотацию типа при проверке на этапе выполнения. Наверное, поэтому `typing.NamedTuple` и `@dataclass` даже не пытаются это делать. Однако некоторые типы, являющиеся конкретными классами, можно использовать в сочетании с `Checked`. К ним относятся простые типы, часто используемые для полей, например `str`, `int`, `float` и `bool`, а также списки значений таких типов.

В примере 24.3 показано, как использовать `Checked` для построения `Movie`.

Пример 24.3. initsub/checkedlib.py: тест создания подкласса `Movie` класса `Checked`

```
>>> class Movie(Checked): ❶
...     title: str ❷
...     year: int
...     box_office: float
...
>>> movie = Movie(title='The Godfather', year=1972, box_office=137) ❸
>>> movie.title
'The Godfather'
>>> movie ❹
Movie(title='The Godfather', year=1972, box_office=137.0)
```

❶ `Movie` наследует классу `Checked`, который будет определен в примере 24.5.
❷ Каждый атрибут аннотируется конструктором. В данном случае я использовал встроенные типы.
❸ При создании экземпляров `Movie` можно задавать только именованные аргументы.
❹ В награду мы получаем удобное представление, созданное методом `__repr__`.

В роли конструктора, используемого в качестве аннотации типа атрибута, может выступать любой вызываемый объект, который принимает нуль или более аргументов и возвращает значение, совместимое с требуемым типом поля, или отвергает аргумент, возбуждая исключение `TypeError` или `ValueError`.

Использование встроенных типов для аннотаций в примере 24.3 означает, что значения должны приниматься конструктором типа. В случае `int` этому условию удовлетворяет любое `x` такое, что `int(x)` возвращает `int`. В случае `str` подойдет вообще любое значение, потому что `str(x)` в Python работает для любого `x`[1].

При вызове без аргументов конструктор должен возвращать значение своего типа по умолчанию[2].

Это стандартное поведение встроенных в Python конструкторов:

```
>>> int(), float(), bool(), str(), list(), dict(), set()
(0, 0.0, False, '', [], {}, set())
```

В подклассах `Checked`, в частности `Movie`, для отсутствующих параметров создаются значения по умолчанию, возвращаемые конструкторами полей. Например:

```
>>> Movie(title='Life of Brian')
Movie(title='Life of Brian', year=0, box_office=0.0)
```

[1] Это справедливо для любого объекта, если только его класс не переопределяет методы `__str__` или `__repr__`, унаследованные от объекта с некорректной реализацией.

[2] При таком решении нельзя будет задавать `None` в качестве значения по умолчанию. Избегать null-значений вообще правильно. Вообще говоря, эта цель труднодостижима, но в некоторых частных случаях не вызывает сложностей. В Python, как и в SQL, я предпочитаю представлять отсутствие данных в текстовом поле пустой строкой, а не `None` или `NULL`. Изучение Go укрепило меня в этой идее: в Go переменные и поля примитивных типов в структурах по умолчанию инициализируются «нулевым значением». См. раздел «Zero values» в онлайновой «Экскурсии по Go» (*https://go.dev/tour/basics/12*), если вам интересно.

Конструкторы используются для проверки во время создания экземпляра и когда атрибуту присваивается значение непосредственно:

```
>>> blockbuster = Movie(title='Avatar', year=2009, box_office='billions')
Traceback (most recent call last):
  ...
TypeError: 'billions' is not compatible with box_office:float
>>> movie.year = 'MCMLXXII'
Traceback (most recent call last):
  ...
TypeError: 'MCMLXXII' is not compatible with year:int
```

Подклассы Checked и статическая проверка типов

При проверке исходного ру-файла, содержащего экземпляр класса `Movie`, определенного в примере 24.3, Муру помечает следующее присваивание как ошибку:

```
movie.year = 'MCMLXXII'
```

Однако Муру не может обнаружить ошибки типизации в таком вызове конструктора:

```
blockbuster = Movie(title='Avatar', year='MMIX')
```

Объясняется это тем, что `Movie` наследует метод `Checked.__init__`, а сигнатура этого метода обязана принимать любые именованные аргументы, чтобы поддержать произвольные определенные пользователем классы.

С другой стороны, если объявить в подклассе `Checked` поле с аннотацией типа `list[float]`, то Муру сможет распознать ошибку при присваивании ему списка с несовместимыми данными, но `Checked` будет игнорировать такой параметр-тип и рассматривать его так же, как `list`.

Теперь рассмотрим реализацию *checkedlib.py*. Начнем с дескрипторного класса `Field`, показанного в примере 24.4.

Пример 24.4. initsub/checkedlib.py: дескрипторный класс `Field`

```
from collections.abc import Callable ❶
from typing import Any, NoReturn, get_type_hints

class Field:
    def __init__(self, name: str, constructor: Callable) -> None: ❷
        if not callable(constructor) or constructor is type(None): ❸
            raise TypeError(f'{name!r} type hint must be callable')
        self.name = name
        self.constructor = constructor

    def __set__(self, instance: Any, value: Any) -> None:
        if value is ...: ❹
            value = self.constructor()
        else:
            try:
                value = self.constructor(value) ❺
            except (TypeError, ValueError) as e: ❻
                type_name = self.constructor.__name__
```

```
                msg = f'{value!r} is not compatible with {self.name}:{type_name}'
                raise TypeError(msg) from e
        instance.__dict__[self.name] = value ❼
```

❶ Напомним, что начиная с Python 3.9 тип `Callable` в аннотациях – это ABC из модуля `collections.abc`, а не объявленный нерекомендуемым тип `typing.Callable`.

❷ Это минимальная аннотация типа `Callable`; параметр-тип и тип возвращаемого значения `constructor` неявно равны `Any`.

❸ Для проверки типа во время выполнения используется встроенная функция `callable`[1]. Проверка на `type(None)` необходима, потому что Python воспринимает `None` в типе как `NoneType`, класс `None` (и потому вызываемый объект), а не как бесполезный конструктор, который только возвращает `None`.

❹ Если `Checked.__init__` присваивает `value` значение `...` (встроенный объект `Ellipsis`), то мы вызываем `constructor` без аргументов.

❺ В противном случае вызываем `constructor` с заданным значением `value`.

❻ Если `constructor` возбуждает одно из этих исключений, то возбуждаем `TypeError` с содержательным сообщением, в котором указаны имена поля и конструктора, например `'MMIX' is not compatible with year:int`.

❼ Если исключений не было, то `value` сохраняется в `instance.__dict__`.

В методе `__set__` мы должны перехватывать исключения `TypeError` и `ValueError`, потому что встроенные конструкторы могут возбуждать любое из них в зависимости от аргумента. Например, `float(None)` возбуждает `TypeError`, но `float('A')` возбуждает `ValueError`. С другой стороны, `float('8')` не возбуждает исключений и возвращает `8.0`. Тем самым я заявляю, что в данном демонстрационном примере это предусмотренное поведение, а не ошибка.

В разделе «LineItem попытка № 4: автоматическое именование атрибутов хранения» главы 23 мы видели, как используется специальный метод `__set_name__` для дескрипторов. В классе `Field` он нам не нужен, потому что экземпляры дескрипторов в исходном коде клиента не создаются; пользователь объявляет типы, являющиеся конструкторами, как мы видели в классе `Movie` (пример 24.3). Что же касается экземпляров дескриптора `Field`, то они создаются во время выполнения методом `Checked.__init_subclass__`, как будет показано в примере 24.5.

Теперь обратимся к классу `Checked`. Я разбил его код на две части. В примере 24.5 показано начало класса, куда вошли наиболее важные методы. Остальные методы показаны в примере 24.6.

Пример 24.5. initsub/checkedlib.py: наиболее важные методы класса `Checked`

```
class Checked:
    @classmethod
    def _fields(cls) -> dict[str, type]: ❶
        return get_type_hints(cls)
```

[1] Я считаю, что `callable` следует сделать допустимым в аннотациях типов. По состоянию на 6 мая 2021 года этот вопрос оставался открытым.

```
    def __init_subclass__(subclass) -> None: ❷
        super().__init_subclass__() ❸
        for name, constructor in subclass._fields().items():   ❹
            setattr(subclass, name, Field(name, constructor)) ❺

    def __init__(self, **kwargs: Any) -> None:
        for name in self._fields():             ❻
            value = kwargs.pop(name, ...)       ❼
            setattr(self, name, value)          ❽
        if kwargs:                              ❾
            self.__flag_unknown_attrs(*kwargs) ❿
```

❶ Этот метод класса я написал, чтобы скрыть вызов `typing.get_type_hints` от остального класса. Если бы нужно было поддерживать только версии Python ⩾ 3.10, то я бы вызвал `inspect.get_annotations`. Проблемы, связанные с этими функциями, описаны в разделе «Проблемы с аннотациями во время выполнения» главы 15.

❷ `__init_subclass__` вызывается, когда определяется подкласс текущего класса. Он получает новый подкласс в первом аргументе, именно поэтому я назвал аргумент `subclass`, а не `cls`, как обычно. Дополнительные сведения по этому вопросу см. во врезке «__init_subclass__ – не обычный метод класса» ниже.

❸ Вызов `super().__init_subclass__()`, строго говоря, не является необходимым, но полезен для правильного взаимодействия с другими классами, которые реализовали `.__init_subclass__()` в том же графе наследования. См. раздел «Множественное наследование и порядок разрешения методов» главы 14.

❹ Обойти все поля `name` и `constructor` ...

❺ ... создавая в `subclass` атрибут с данным именем `name`, связанный с дескриптором `Field` с параметрами `name` и `constructor`.

❻ Для каждого поля класса `name` ...

❼ ... получить соответствующее значение `value` из `kwargs` и удалить его из `kwargs`. Использование `...` (объект `Ellipsis`) в качестве значения по умолчанию позволяет отличить заданные аргументы со значением `None` от незаданных[1].

❽ Этот вызов `setattr` вызывает срабатывание метода `Checked.__setattr__`, показанного в примере 24.6.

❾ Если в `kwargs` остались аргументы, то их имена не совпадают ни с одним из объявленных полей, и `__init__` завершается ошибкой.

❿ Об этой ошибке сообщает метод `__flag_unknown_attrs`, показанный в примере 24.6. Он принимает аргумент `*names`, содержащий имена неизвестных атрибутов. Я использовал в `*kwargs` одну звездочку, чтобы передать ключи в виде последовательности аргументов.

[1] Как отмечалось в разделе «Циклы, сигнальные маркеры и отравленные таблетки» главы 19, объект `Ellipsis` – удобный и безопасный сигнальный маркер. Он существует уже давно, но в последнее время ему нашлись новые применения, например в аннотациях типов и в NumPy.

__init_subclass__ – не обычный метод класса

Декоратор `@classmethod` никогда не применяется к методу `__init_subclass__`, но это не так важно, потому что специальный метод `__new__` ведет себя как метод класса даже без `@classmethod`. Первый аргумент, который Python передает методу `__init_subclass__`, – это класс. Но отнюдь не класс, в котором реализован `__init_subclass__`, а вновь определенный подкласс этого класса. Это непохоже на `__new__` и на все остальные известные мне методы класса. Поэтому я считаю, что `__init_subclass__` не является методом класса в обычном смысле слова и называть его первый аргумент `cls` было бы неразумно и только сбивало бы с толку. В документации по `__init_suclass__` (*https://docs.python.org/3/reference/datamodel.html#object.__init_subclass__*) аргумент назван `cls`, но далее приводится пояснение: «… вызывается, когда создается подкласс объемлющего класса. В этот момент `cls` – новый подкласс».

Теперь рассмотрим остальные методы класса `Checked`. Имена методов `_fields` и `_asdict` начинаются знаком подчеркивания по той же причине, что в API класса `collections.namedtuple`: чтобы уменьшить вероятность конфликта с именами определенных пользователем полей.

Пример 24.6. initsub/checkedlib.py: остальные методы класса `Checked`

```
    def __setattr__(self, name: str, value: Any) -> None: ❶
        if name in self._fields(): ❷
            cls = self.__class__
            descriptor = getattr(cls, name)
            descriptor.__set__(self, value) ❸
        else:
            self.__flag_unknown_attrs(name)  ❹

    def __flag_unknown_attrs(self, *names: str) -> NoReturn: ❺
        plural = 's' if len(names) > 1 else ''
        extra = ', '.join(f'{name!r}' for name in names)
        cls_name = repr(self.__class__.__name__)
        raise AttributeError(f'{cls_name} object has no attribute{plural} {extra}')

    def _asdict(self) -> dict[str, Any]: ❻
        return {
            name: getattr(self, name)
            for name, attr in self.__class__.__dict__.items()
            if isinstance(attr, Field)
        }

    def __repr__(self) -> str: ❼
        kwargs = ', '.join(
            f'{key}={value!r}' for key, value in self._asdict().items()
        )
        return f'{self.__class__.__name__}({kwargs})'
```

❶ Перехватывать все попытки установить атрибут экземпляра. Необходимо, чтобы предотвратить присваивание неизвестному атрибуту.

❷ Если атрибут с именем `name` известен, получить соответствующий дескриптор.

❸ Обычно нам нет нужды вызывать метод `__set__` дескриптора явно. Но в данном случае это необходимо, потому что `__setattr__` перехватывает все попытки установить атрибут экземпляра, даже при наличии переопределяющего дескриптора типа `Field`[1].

❹ В противном случае атрибут с именем `name` неизвестен, и метод `__flag_unknown_attrs` возбуждает исключение.

❺ Сконструировать полезное сообщение об ошибке, содержащее все неожиданные аргументы, и возбудить исключение `AttributeError`. Это редкий пример специального типа `NoReturn`, рассмотренного в одноименном разделе главы 8.

❻ Создать словарь, содержащий атрибуты объекта `Movie`. Я предпочел бы назвать этот метод `_as_dict`, но решил последовать соглашению, заложенному методом `_asdict` в классе `collections.namedtuple`.

❼ Желание реализовать полезное представление в методе `__repr__` – основная причина наличия метода `_asdict` в этом примере.

В примере `Checked` показано, как быть с переопределяющими дескрипторами при реализации метода `__setattr__`, чтобы блокировать попытки присвоить значение произвольному атрибуту после создания экземпляра. Стоит ли реализовывать `__setattr__` в этом примере – спорный вопрос. Без него присваивание `movie.director = 'Greta Gerwig'` прошло бы успешно, но атрибут `director` в любом случае не проверяется, поэтому он не появился бы в `__repr__` и не был бы включен в словарь, возвращаемый методом `_asdict` (оба этих метода определены в примере 24.6).

В скрипте *record_factory.py* (пример 24.2) я решил эту проблему с помощью атрибута класса `__slots__`. Однако в данном случае это более простое решение не годится по причине, объясняемой ниже.

Почему __init_subclass__ не может конфигурировать __slots__

Атрибут `__slots__` дает эффект, только если это один из элементов в пространстве имен класса, передаваемого `type.__new__`. Добавление `__slots__` в уже существующий класс не дает ничего. Python вызывает `__init_subclass__` только после того, как класс уже построен – слишком поздно для конфигурирования `__slots__`. Декоратор класса тоже не может сконфигурировать `__slots__`, потому что применяется еще позже, чем `__init_subclass__`. Мы рассмотрим эти вопросы в разделе «Что когда происходит: этап импорта и этап выполнения» ниже.

Чтобы сконфигурировать `__slots__` во время выполнения, ваш код должен самостоятельно построить пространство имен класса, передаваемое в качестве последнего аргумента методу `type.__new__`. Для этого можно написать фабричную функцию наподобие той, что была показана в скрипте *record_factory.py*, или пойти на крайние меры и реализовать метакласс. Как динамически конфигурировать `__slots__`, мы увидим в разделе «Основы метаклассов» ниже.

До того, как в документе PEP 487 (*https://peps.python.org/pep-0487/*) было предложено упростить настройку создания класса с помощью метода `__init_subclass__`, а в версии Python 3.7 это предложение было реализовано, подобную

[1] Тонкая концепция переопределяющего дескриптора обсуждалась в главе 23.

функциональность приходилось реализовывать с помощью декоратора классов. Это тема следующего раздела.

Дополнение класса с помощью декоратора класса

Декоратор класса – это вызываемый объект, который ведет себя как декоратор функции: получает декорированный класс в качестве аргумента и возвращает заменяющий его класс. Часто декораторы классов возвращают сам декорированный класс после внедрения в него новых методов с помощью присваивания атрибутам.

Пожалуй, самая распространенная причина предпочесть декоратор класса более простому методу `__init_subclass__` – желание избежать интерференции с другими возможностями классов, в т. ч. наследованием и метаклассами[1].

В этом разделе мы изучим скрипт *checkeddeco.py*, который представляет ту же функциональность, что *checkedlib.py*, но с помощью декоратора класса. Как обычно, начнем с примера использования, взятого из тестов в файле *checkeddeco.py* (пример 24.7).

Пример 24.7. checkeddeco.py: создание класса `Movie`, снабженного декоратором `@checked`

```
>>> @checked
... class Movie:
...     title: str
...     year: int
...     box_office: float
...
>>> movie = Movie(title='The Godfather', year=1972, box_office=137)
>>> movie.title
'The Godfather'
>>> movie
Movie(title='The Godfather', year=1972, box_office=137.0)
```

Единственное различие между примерами 24.7 и 24.3 – способ объявления класса `Movie`: он снабжен декоратором `@checked`, а не наследует `Checked`. Со стороны поведение обеих версий выглядит одинаково, включая проверку типов и присваивание значений по умолчанию, показанные после примера 24.3 в разделе «Введение в __init_subclass__».

Теперь обратимся к реализации *checkeddeco.py*. Предложения импорта и класс `Field` такие же, как в *checkedlib.py* (пример 24.4). В *checkeddeco.py* нет больше никаких классов, только функции.

Логика, которая раньше была реализована в `__init_subclass__`, теперь перенесена в функцию `checked` – декоратор класса, показанный в примере 24.8.

Пример 24.8. checkeddeco.py: декоратор класса

```
def checked(cls: type) -> type: ❶
    for name, constructor in _fields(cls).items(): ❷
        setattr(cls, name, Field(name, constructor)) ❸
```

[1] Это обоснование выдвигается в реферате документа PEP 557 «Data Classes» (*https://peps.python.org/pep-0557/#abstract*), чтобы объяснить, почему была выбрана реализация в виде декоратора класса.

```
    cls._fields = classmethod(_fields) # type: ignore ❹

    instance_methods = ( ❺
        __init__,
        __repr__,
        __setattr__,
        _asdict,
        __flag_unknown_attrs,
    )
    for method in instance_methods: ❻
        setattr(cls, method.__name__, method)

    return cls ❼
```

❶ Напомним, что классы – это экземпляры `type`. Эти аннотации типов позволяют предположить, что мы имеем дело с декоратором класса: он принимает и возвращает класс.
❷ `_fields` – функция верхнего уровня, определенная в модуле позднее (в примере 24.9).
❸ Замена каждого атрибута, возвращенного `_fields`, экземпляром дескриптора `Field` – то, что делал метод `__init_subclass__` в примере 24.5. Здесь нам предстоит больше работы…
❹ Построить метод класса по `_fields` и добавить его в декорированный класс. Комментарий `type: ignore` необходим, т. к. Муру ругается, что в `type` нет атрибута `_fields`.
❺ Функции уровня модуля станут методами экземпляра в декорированном классе.
❻ Добавить все элементы `instance_methods` в `cls`.
❼ Вернуть декорированный `cls`, выполнив основной контракт декоратора класса.

Имена всех функций верхнего уровня в *checkeddeco.py* начинаются знаком подчеркивания, кроме декоратора `checked`. Это соглашение об именовании имеет смысл по двум причинам:

- `checked` является частью открытого интерфейса модуля *checkeddeco.py*, а остальные функции – нет;
- функции из примера 24.9 будут внедрены в декорированный класс, и префикс `_` уменьшает вероятность конфликтов имени с определенными пользователем атрибутами и методами этого класса.

Оставшаяся часть *checkeddeco.py* приведена в примере 24.9. Код этих функций верхнего уровня такой же, как у соответственных методов класса `Checked` из файла *checkedlib.py*. Все пояснения были даны в примерах 24.5 и 24.6.

Отметим, что у функции `_fields` двоякая роль в файле *checkeddeco.py*. В первой строке декоратора `checked` она используется как обычная функция, а затем внедряется в качестве метода декорированного класса.

Пример 24.9. checkeddeco.py: методы, внедряемые в декорированный класс

```
def _fields(cls: type) -> dict[str, type]:
    return get_type_hints(cls)
```

```
def __init__(self: Any, **kwargs: Any) -> None:
    for name in self._fields():
        value = kwargs.pop(name, ...)
        setattr(self, name, value)
    if kwargs:
        self.__flag_unknown_attrs(*kwargs)

def __setattr__(self: Any, name: str, value: Any) -> None:
    if name in self._fields():
        cls = self.__class__
        descriptor = getattr(cls, name)
        descriptor.__set__(self, value)
    else:
        self.__flag_unknown_attrs(name)

def __flag_unknown_attrs(self: Any, *names: str) -> NoReturn:
    plural = 's' if len(names) > 1 else ''
    extra = ', '.join(f'{name!r}' for name in names)
    cls_name = repr(self.__class__.__name__)
    raise AttributeError(f'{cls_name} has no attribute{plural} {extra}')

def _asdict(self: Any) -> dict[str, Any]:
    return {
        name: getattr(self, name)
        for name, attr in self.__class__.__dict__.items()
        if isinstance(attr, Field)
    }

def __repr__(self: Any) -> str:
    kwargs = ', '.join(
        f'{key}={value!r}' for key, value in self._asdict().items()
    )
    return f'{self.__class__.__name__}({kwargs})'
```

Модуль *checkeddeco.py* реализует простой, но полезный декоратор класса. Декоратор Python `@dataclass` делает гораздо больше. Он поддерживает много конфигурационных параметров, добавляет в декорированный класс больше методов, обрабатывает конфликты с пользовательскими методами декорированного класса или предупреждает о них и даже обходит `__mro__`, чтобы собрать определенные пользователем атрибуты, объявленные в суперклассах. Исходный код (*https://github.com/python/cpython/blob/3.9/Lib/dataclasses.py*) пакета `dataclasses` в версии Python 3.9 занимает больше 1200 строк.

Если мы хотим заниматься метапрограммированием классов, то должны знать, когда интерпретатор Python обрабатывает каждый блок кода при построении класса. Этот вопрос рассматривается далее.

Что когда происходит: этап импорта и этап выполнения

Программисты на Python употребляют термины «этап импорта» и «этап выполнения», но они определены нестрого, так что между ними существует ничейная земля.

На этапе импорта интерпретатор:

1) производит синтаксический анализ исходного кода *ру*-модуля сверху вниз. Именно в это время могут возникать исключения `SyntaxError`;
2) генерирует исполняемый байт-код;
3) выполняет код верхнего уровня откомпилированного модуля.

Если в локальном кеше `__pycache__` существует актуальный *рус*-файл, то этот этап пропускается, поскольку уже имеется готовый к выполнению байт-код.

Хотя синтаксический анализ и компиляция, безусловно, являются действиями, выполняемыми на «этапе импорта», на этой стадии могут происходить и другие вещи, потому что почти каждое предложение в Python является исполняемым в том смысле, что в нем может выполняться пользовательский код, изменяющий состояние программы.

В частности, предложение `import` – не просто объявление[1], оно еще и выполняет весь код, находящийся на верхнем уровне импортируемого модуля, при первом его импорте в память процесса. При последующих операциях импорта того же модуля используется кешированный код, так что происходит только связывание имен. Этот верхнеуровневый код может делать все, что угодно, включая такие типичные для «этапа выполнения» действия, как подключение к базе данных[2]. Потому-то граница между «этапом импорта» и «этапом выполнения» размыта: предложение `import` может активировать любые действия, которые принято считать частью «этапа выполнения». И наоборот, «этап импорта» может случиться глубоко внутри этапа выполнения, потому что предложение `import` и встроенная функция `__import__` могут употребляться внутри любой обычной функции.

Все это довольно тонкие и абстрактные материи, поэтому ниже описано несколько экспериментов, которые помогут разобраться, что когда происходит.

Демонстрация работы интерпретатора

Рассмотрим скрипт *evaldemo.py*, в котором используется декоратор класса, дескриптор и построитель класса, основанный на методе `__init_subclass__`. Все они определены в модуле *builderlib.py*. В модулях есть несколько вызовов `print`, которые показывают, что происходит под капотом. Никаких других полезных функций они не несут. Цель экспериментов – узнать, в каком порядке выполняются эти вызовы `print`.

Применение и декоратора, и построителя класса к классу, в котором определен метод `__init_subclass__`, скорее всего, является признаком зауми или отчаяния. Но эта необычная комбинация полезна, чтобы показать, в какой момент применяются декоратор класса и `__init_subclass__`.

Начнем с изучения скрипта *builderlib.py*, разбив его на две части: примеры 24.10 и 24.11.

---

[1] В отличие от предложения `import` в Java, которое служит только объявлением, извещающим компилятор о необходимости загрузить некоторые пакеты.

[2] Я не хочу сказать, что подключение к базе данных по самому факту импорта модуля – хорошая идея, а лишь отмечаю, что это возможно.

Пример 24.10. builderlib.py: начало модуля

```
print('@ builderlib module start')

class Builder: ❶
    print('@ Builder body')

    def __init_subclass__(cls): ❷
        print(f'@ Builder.__init_subclass__({cls!r})')

        def inner_0(self):
            print(f'@ SuperA.__init_subclass__:inner_0({self!r})')

        cls.method_a = inner_0

    def __init__(self): ❸
        super().__init__()
        print(f'@ Builder.__init__({self!r})')

    def deco(cls): ❹
        print(f'@ deco({cls!r})')

        def inner_1(self): ❺
            print(f'@ deco:inner_1({self!r})')

        cls.method_b = inner_1
        return cls ❻
```

❶ Это построитель классов, в котором будет реализован ...
❷ ... метод `__init_subclass__`.
❸ Определить функцию, добавляемую в подкласс в присваивании ниже.
❹ Декоратор класса.
❺ Функция, добавляемая в декорированный класс.
❻ Вернуть класс, полученный в качестве аргумента.

Продолжаем файл *builderlib.py*, начатый в примере 24.11...

Пример 24.11. builderlib.py: конец модуля

```
class Descriptor: ❶
    print('@ Descriptor body')

    def __init__(self): ❷
        print(f'@ Descriptor.__init__({self!r})')

    def __set_name__(self, owner, name): ❸
        args = (self, owner, name)
        print(f'@ Descriptor.__set_name__{args!r}')

    def __set__(self, instance, value): ❹
        args = (self, instance, value)
        print(f'@ Descriptor.__set__{args!r}')

    def __repr__(self):
        return '<Descriptor instance>'

print('@ builderlib module end')
```

❶ Дескрипторный класс, демонстрирующий, когда …
❷ … создается экземпляр дескриптора и когда …
❸ … вызывается метод `__set_name__` при конструировании класса `owner`.
❹ Как и все прочие методы, этот метод `__set__` только распечатывает свои аргументы и больше ничего не делает.

Импортировав *builderlib.py* в консольном сеансе, мы увидим вот что:

```
>>> import builderlib
@ builderlib module start
@ Builder body
@ Descriptor body
@ builderlib module end
```

Отметим, что строки, напечатанные *builderlib.py*, начинаются символом `@`.

Теперь займемся файлом *evaldemo.py*, который активирует специальные методы в *builderlib.py* (пример 24.12).

Пример 24.12. evaldemo.py: скрипт для экспериментов с builderlib.py

```
#!/usr/bin/env python3

from builderlib import Builder, deco, Descriptor

print('# evaldemo module start')

@deco ❶
class Klass(Builder): ❷
    print('# Klass body')

    attr = Descriptor() ❸

    def __init__(self):
        super().__init__()
        print(f'# Klass.__init__({self!r})')

    def __repr__(self):
        return '<Klass instance>'

def main(): ❹
    obj = Klass()
    obj.method_a()
    obj.method_b()
    obj.attr = 999

if __name__ == '__main__':
    main()

print('# evaldemo module end')
```

❶ Применить декоратор.
❷ Унаследовать `Builder`, чтобы активировать его метод `__init_subclass__`.
❸ Создать экземпляр дескриптора.
❹ Вызывается, только если модуль запущен как главная программа.

В вызовы `print` в `evaldemo.py` включен префикс `#`. Снова открыв консоль и импортировав *evaldemo.py*, мы увидим строки, показанные в примере 24.13.

Пример 24.13. Эксперимент с evaldemo.py на консоли

```
>>> import evaldemo
@ builderlib module start❶
@ Builder body
@ Descriptor body
@ builderlib module end
# evaldemo module start
# Klass body ❷
@ Descriptor.__init__(<Descriptor instance>) ❸
@ Descriptor.__set_name__(<Descriptor instance>,
    <class 'evaldemo.Klass'>, 'attr') ❹
@ Builder.__init_subclass__(<class 'evaldemo.Klass'>) ❺
@ deco(<class 'evaldemo.Klass'>) ❻
# evaldemo module end
```

❶ Первые четыре строки – результат выполнения `from builderlib import…`. Их не будет, если вы не закрывали консоль после предыдущего эксперимента, потому что тогда *builderlib.py* уже загружен.

❷ Сигнал о том, что интерпретатор Python начал читать тело `Klass`. В этой точке объект класса еще не существует.

❸ Экземпляр дескриптора создан и связан с `attr` в пространстве имен, которое Python передает конструктору объект класса по умолчанию: `type.__new__`.

❹ В этой точке Python встроенный метод `type.__new__` создал объект `Klass` и вызывает метод `__set_name__` каждого экземпляра дескрипторных классов, предоставляющих такой метод, передавая `Klass` в качестве аргумента `owner`.

❺ Затем `type.__new__` вызывает метод `__init_subclass__` суперкласса `Klass`, передавая `Klass` в качестве единственного аргумента.

❻ Когда `type.__new__` возвращает объект класса, Python применяет декоратор. В этом примере класс, возвращенный `deco`, связывается с именем `Klass` в пространстве имен модуля.

Метод `type.__new__` написан на C. Описанное выше поведение документировано в разделе «Создание объекта класса» (*https://docs.python.org/3/reference/datamodel.html#creating-the-class-object*) секции «Модель данных» справочного руководства по Python.

Обратите внимание, что функция `main()` скрипта *evaldemo.py* в этом консольном сеансе не вычислялась, поэтому экземпляр класса `Klass` не создавался. Все, что мы видим, – результат операций на «этапе импорта»: импортирования модуля `builderlib` и определения `Klass`.

Запустив *evaldemo.py* как скрипт, мы увидим те же строки, что в примере 24.13, плюс дополнительные в конце распечатки. Эти дополнительные строки – результат выполнения `main()` (пример 24.14).

Пример 24.14. Запуск evaldemo.py как программы

```
$ ./evaldemo.py
[... 9 строк опущено ...]
@ deco(<class '__main__.Klass'>) ❶
@ Builder.__init__(<Klass instance>) ❷
# Klass.__init__(<Klass instance>)
@ SuperA.__init_subclass__:inner_0(<Klass instance>) ❸
@ deco:inner_1(<Klass instance>) ❹
```

```
@ Descriptor.__set__(<Descriptor instance>, <Klass instance>, 999) ❺
# evaldemo module end
```

❶ Первые 10 строк, включая эту, такие же, как в примере 24.13.
❷ Напечатана в результате выполнения `super().__init__()` в `Klass.__init__`.
❸ Напечатана `obj.method_a()` в `main`; `method_a` был внедрен методом `SuperA.__init_subclass__`.
❹ Напечатана `obj.method_b()` в `main`; `method_b` был внедрен декоратором `deco`.
❺ Напечатана в результате выполнения `obj.attr = 999` в `main`.

Базовый класс с методом `__init_subclass__` и декоратор класса – мощные инструменты, но они могут работать только с классом, который уже создан методом `type.__new__` где-то в недрах интерпретатора. В редких случаях, когда требуется подправить аргументы, передаваемые `type.__new__`, нам необходим метакласс. И это последний вопрос, рассматриваемый в этой главе – и вообще в книге.

Основы метаклассов

> Магия метаклассов не интересна 99 % пользователей. Если вы задаетесь вопросом, нужны ли они вам, значит, не нужны (люди, которым они нужны, точно знают, что нуждаются в них, и не нуждаются в объяснениях).
>
> – Тим Питерс, изобретатель алгоритма timsort и плодовитый программист на Python[1]

Метакласс – это фабрика классов, но, в отличие от функции наподобие `record_factory` из примера 24.2, метакласс записывается в виде класса. Иными словами, метакласс – это класс, экземплярами которого являются классы. На рис. 24.1 изображен метакласс в нотации хреновин и штуковин: одна хреновина порождает другую.

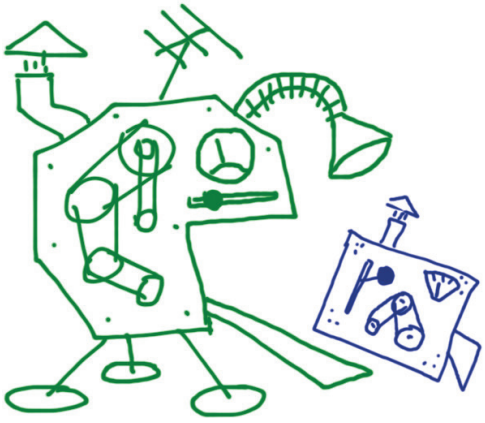


Рис. 24.1. Метакласс – это класс, который создает классы

[1] Сообщение в группе comp.lang.python, озаглавленное: «Acrimony in c.l.p». (*https://mail.python.org/pipermail/python-list/2002-December/134521.html*). Это вторая часть сообщения от 23 декабря 2002, процитированного в предисловии. В тот день на TimBot, видно, напало вдохновение.

В объектной модели Python классы являются объектами, поэтому каждый класс должен быть экземпляром какого-то другого класса. По умолчанию классы Python являются экземплярами класса `type`. Иными словами, `type` – метакласс для большинства встроенных и пользовательских классов:

```
>>> str.__class__
<class 'type'>
>>> from bulkfood_v5 import LineItem
>>> LineItem.__class__
<class 'type'>
>>> type.__class__
<class 'type'>
```

Чтобы избежать бесконечного спуска, `type` является экземпляром себя самого, как видно из последней строки.

Обратите внимание: я не говорю, что `str` или `LineItem` наследуют классу `type`. Я утверждаю, что `str` и `LineItem` – экземпляры `type`. И все они являются подклассами `object`. Возможно, рис. 24.2 поможет вам освоиться в этой странной реальности.

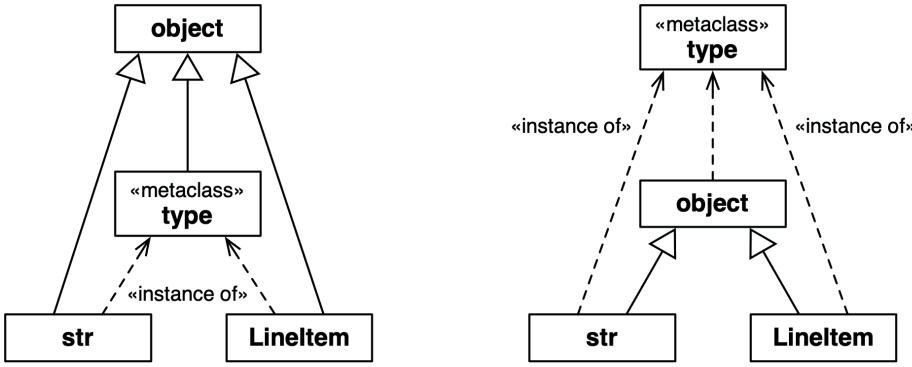


Рис. 24.2. Обе диаграммы правильны. На левой показано, что `str`, `type` и `LineItem` – подклассы `object`. Из правой видно, что `str`, `object` и `LineItem` – экземпляры `type`, поскольку все они – классы

Между классами `object` и `type` имеется удивительная связь: `object` – экземпляр `type`, а `type` – подкласс `object`. Эта связь «магическая»: выразить ее средствами Python невозможно, потому что любой из этих классов должен существовать, прежде чем можно будет определить другой. И тот факт, что `type` является экземпляром самого себя, – тоже магия.

В следующем фрагменте кода показано, что классом `collections.Iterable` является `abc.ABCMeta`. Класс `Iterable` абстрактный, а `ABCMeta` – нет; да и как может быть иначе, если `Iterable` является экземпляром `ABCMeta`:

```
>>> from collections.abc import Iterable
>>> Iterable.__class__
<class 'abc.ABCMeta'>
>>> import abc
>>> from abc import ABCMeta
>>> ABCMeta.__class__
<class 'type'>
```

Классом `ABCMeta` также является `type`. Любой класс является экземпляром `type`, прямо или косвенно, но только метаклассы являются также подклассами `type`. Это самое главное, что нужно знать о метаклассах: любой метакласс, в частности `ABCMeta`, наследует от `type` могущество, необходимое для конструирования классов. На рис. 24.3 показана эта важнейшая связь.

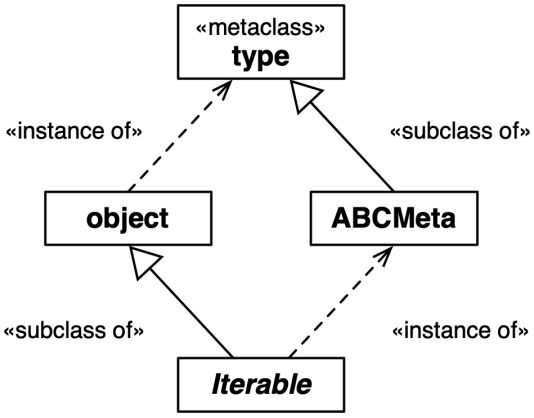


Рис. 24.3. `Iterable` – подкласс `object` и экземпляр `ABCMeta`. И `object`, и `ABCMeta` – экземпляры `type`, но ключевая связь здесь – тот факт, что `ABCMeta` – еще и подкласс `type`, поскольку `ABCMeta` является метаклассом. На этой диаграмме `Iterable` – единственный абстрактный класс

Необходимо твердо запомнить, что все классы являются экземплярами `type`, и именно поэтому они работают как фабрики классов. Метакласс может настраивать экземпляры посредством реализации специальных методов, как показано в следующих разделах.

Как метакласс настраивает класс

Чтобы использовать метакласс, важно понимать, как метод `__new__` применяется к любому классу. Этот вопрос мы обсуждали в разделе «Гибкое создание объектов с помощью метода __new__» главы 22.

Тот же механизм работает на «метауровне», когда метакласс собирается создать новый экземпляр, т. е. класс. Рассмотрим следующее объявление:

```
class Klass(SuperKlass, metaclass=MetaKlass):
    x = 42
    def __init__(self, y):
        self.y = y
```

Чтобы обработать предложение `class`, Python вызывает метод `MetaKlass.__new__` с такими аргументами:

`meta_cls`

Сам метакласс (`MetaKlass`), потому что `__new__` работает как метод класса.

`cls_name`

Строка `Klass`.

`bases`

Одноэлементный кортеж (`SuperKlass,`), который может иметь больше элементов в случае множественного наследования.

`cls_dict`

Отображение вида:

```
{x: 42, `__init__`: <function __init__ at 0x1009c4040>}
```

При реализации `MetaKlass.__new__` мы можем проинспектировать и изменить эти аргументы до передачи их методу `super().__new__`, который в конечном итоге вызовет `type.__new__` для создания нового объекта класса.

После возврата из `super().__new__` мы можем дообработать вновь созданный класс, перед тем как возвращать его Python. Затем интерпретатор вызывает `SuperKlass.__init_subclass__`, передавая созданный нами класс, после чего применяет к нему декоратор класса, если таковой задан. Наконец, Python связывает объект класса с его именем в объемлющем пространстве имен – обычно это глобальное пространство имен модуля, если предложение `class` находится на верхнем уровне.

Чаще всего в методе метакласса `__new__` добавляются или заменяются элементы отображения `cls_dict`, которое представляет пространство имен конструируемого класса. Например, перед вызовом `super().__new__` мы можем внедрить методы в конструируемый класс, добавив функции в `cls_dict`. Заметим, однако, что добавлять методы можно и после построения класса, именно это и делают `__init_subclass__` или декоратор класса.

Атрибут, который должен быть добавлен в `cls_dict` до вызова `type.__new__`, – `__slots__`; этот вопрос обсуждался в разделе «Почему __init_subclass__ не может конфигурировать __slots__» выше. Метод `__new__` метакласса – идеальное место для конфигурирования `__slots__`. В следующем разделе объясняется, как это сделать.

Элегантный пример метакласса

Представленный ниже метакласс `MetaBunch` – вариация на тему последнего примера в главе 4 книги Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», 3-е издание, работающего в Python 2.7 и 3.5[1]. Для версии Python 3.6 и более поздних я смог еще упростить код.

Сначала посмотрим, что предоставляет базовый класс `Bunch`:

```
>>> class Point(Bunch):
...     x = 0.0
...     y = 0.0
...     color = 'gray'
...
```

[1] Авторы любезно разрешили мне воспользоваться их примером. MetaBunch впервые упоминается в сообщении, опубликованном Мартелли в группе 7 июля 2002 года, с заголовком «элегантный пример метакласса (было Re: структуры в python)» (*https://mail.python.org/pipermail/python-list/2002-July/162558.html*), после обсуждения записеподобных структур данных в Python. Оригинальный код Мартелли для Python 2.2 по-прежнему работает после одного изменения: чтобы использовать метакласс в Python 3, необходимо включить именованный аргумент `metaclass` в объявление класса, например `Bunch(metaclass=MetaBunch)`, – вместо старого соглашения, требовавшего добавлять атрибут уровня класса `__metaclass__`.

```
>>> Point(x=1.2, y=3, color='green')
Point(x=1.2, y=3, color='green')
>>> p = Point()
>>> p.x, p.y, p.color
(0.0, 0.0, 'gray')
>>> p
Point()
```

Напомним, что `Checked` назначает имена дескрипторам `Field` в подклассах, исходя из аннотаций типов в переменных класса, которые не становятся атрибутами класса, поскольку не имеют значений.

С другой стороны, в подклассах `Bunch` используются фактические атрибуты класса со значениями, которые затем становятся значениями атрибутов экземпляра по умолчанию. Сгенерированный метод `__repr__` опускает аргументы для атрибутов, принимающих значения по умолчанию.

`MetaBunch` – метакласс `Bunch` – генерирует атрибут `__slots__` для нового класса на основе атрибутов класса, объявленных в пользовательском классе. Это блокирует создание и последующее присваивание необъявленным атрибутам:

```
>>> Point(x=1, y=2, z=3)
Traceback (most recent call last):
  ...
AttributeError: No slots left for: 'z'
>>> p = Point(x=21)
>>> p.y = 42
>>> p
Point(x=21, y=42)
>>> p.flavor = 'banana'
Traceback (most recent call last):
  ...
AttributeError: 'Point' object has no attribute 'flavor'
```

А теперь перейдем к элегантному коду `MetaBunch`.

Пример 24.15. metabunch/from3.6/bunch.py: метакласс `MetaBunch` и класс `Bunch`

```
class MetaBunch(type): ❶
    def __new__(meta_cls, cls_name, bases, cls_dict): ❷

        defaults = {} ❸

        def __init__(self, **kwargs): ❹
            for name, default in defaults.items(): ❺
                setattr(self, name, kwargs.pop(name, default))
            if kwargs: ❻
                extra = ', '.join(kwargs)
                raise AttributeError(f'No slots left for: {extra!r}')

        def __repr__(self): ❼
            rep = ', '.join(f'{name}={value!r}'
                            for name, default in defaults.items()
                            if (value := getattr(self, name)) != default)
            return f'{cls_name}({rep})'

        new_dict = dict(__slots__=[], __init__=__init__, __repr__=__repr__) ❽
```

```
        for name, value in cls_dict.items(): ❾
            if name.startswith('__') and name.endswith('__'): ❿
                if name in new_dict:
                    raise AttributeError(f"Can't set {name!r} in {cls_name!r}")
                new_dict[name] = value
            else: ⓫
                new_dict['__slots__'].append(name)
                defaults[name] = value
        return super().__new__(meta_cls, cls_name, bases, new_dict) ⓬

class Bunch(metaclass=MetaBunch): ⓭
    pass
```

❶ Для создания нового метакласса унаследовать `type`.

❷ `__new__` работает, как метод класса, но класс является метаклассом, поэтому я назвал первый аргумент `meta_cls` (часто употребляют также имя `mcs`). Остальные три аргумента такие же, как в трехаргументной сигнатуре для вызова `type()` с целью непосредственного создания класса.

❸ В `defaults` будет храниться отображение имен атрибутов на их значения по умолчанию.

❹ Этот метод будет внедрен в новый класс.

❺ Прочитать `defaults` и присвоить соответствующему атрибуту экземпляра значение, извлеченное из `kwargs` или подразумеваемое по умолчанию.

❻ Если в `kwargs` остались аргументы, значит, не нашлось слотов, в которые их можно было бы поместить. Мы полагаем, что *быстрый отказ* – правильный подход, поэтому не хотим молчаливо игнорировать лишние элементы. Быстрое и эффективное решение – выбирать элементы из `kwargs` по одному и пытаться установить их в экземпляре, а если не получается, сразу возбуждать исключение `AttributeError`.

❼ `__repr__` возвращает строку, которая выглядит как вызов конструктора, например `Point(x=3)`. При этом именованные аргументы, принимающие значения по умолчанию, опускаются.

❽ Инициализировать пространство имен для нового класса.

❾ Обойти пространство имен пользовательского класса.

❿ Если найдено имя `name` с двумя подчерками, копировать элемент в пространство имен нового класса, если его там еще нет. Это не даст пользователю перезаписать `__init__`, `__repr__` и другие атрибуты, установленные самим Python, например `__qualname__` и `__module__`.

⓫ Если имя атрибута `name` не начинается двумя подчерками, добавить его в конец `__slots__` и сохранить его значение `value` в `defaults`.

⓬ Построить и вернуть новый класс.

⓭ Предоставить базовый класс, чтобы пользователи не видели `MetaBunch`.

`MetaBunch` работает, потому что сконфигурировал `__slots__`, перед тем как вызывать `super().__new__` для построения окончательного класса. Как всегда при метапрограммировании, важно понимать последовательность действий. Проведем еще один эксперимент для демонстрации работы интерпретатора, на сей раз с метаклассом.

Демонстрация работы метакласса

Это вариация на тему раздела «Демонстрация работы интерпретатора» выше, но теперь мы добавили метакласс. Модуль *builderlib.py* такой же, как и раньше, а главный скрипт *evaldemo_meta.py* показан в примере 24.16.

Пример 24.16. evaldemo_meta.py: эксперимент с метаклассом

```
#!/usr/bin/env python3

from builderlib import Builder, deco, Descriptor
from metalib import MetaKlass ❶

print('# evaldemo_meta module start')
@deco
class Klass(Builder, metaclass=MetaKlass): ❷
    print('# Klass body')

    attr = Descriptor()

    def __init__(self):
        super().__init__()
        print(f'# Klass.__init__({self!r})')

    def __repr__(self):
        return '<Klass instance>'

def main():
    obj = Klass()
    obj.method_a()
    obj.method_b()
    obj.method_c() ❸
    obj.attr = 999

if __name__ == '__main__':
    main()

print('# evaldemo_meta module end')
```

❶ Импортировать `MetaKlass` из модуля *metalib.py*, который будет показан в примере 24.18.
❷ Объявить `Klass` как подкласс `Builder` и экземпляр `MetaKlass`.
❸ Этот метод внедряется `MetaKlass.__new__`, как мы увидим ниже.

В научных целях я в примере 24.16 вопреки всем резонам применил к `Klass` сразу три техники метапрограммирования: декоратор, базовый класс с методом `__init_subclass__` и специализированный метакласс. Если вам придет в голову сделать это в реальной программе, пожалуйста, не ссылайтесь на меня. Повторю: моей целью было показать место каждой техники в процессе конструирования класса и их взаимное влияние.

Как и в предыдущем эксперименте, код не делает ничего полезного, а только печатает сообщения, раскрывающие ход выполнения. В примере 24.17 показан код первой части *metalib.py*, а остаток – в примере 24.18.

Пример 24.17. metalib.py: класс `NosyDict`

```
print('% metalib module start')

import collections

class NosyDict(collections.UserDict):
    def __setitem__(self, key, value):
        args = (self, key, value)
        print(f'% NosyDict.__setitem__{args!r}')
        super().__setitem__(key, value)

    def __repr__(self):
        return '<NosyDict instance>'
```

В классе `NosyDict` я переопределил метод `__setitem__`, так чтобы показывать устанавливаемые ключи `key` и значения `value`. Метакласс будет использовать экземпляр `NosyDict` для хранения пространства имен конструируемого класса, что позволит нам еще лучше понять, как работает интерпретатор Python.

В *metalib.py* основной интерес представляет метакласс, показанный в примере 24.18. Он реализует специальный метод класса `__prepare__`, который Python вызывает только для метаклассов. Метод `__prepare__` – это самая ранняя возможность повлиять на процесс создания нового класса.

При написании метакласса я стараюсь придерживаться следующего соглашения об именовании аргументов специального метода:

- использовать `cls` вместо `self` для методов экземпляра, потому что экземпляр является классом;
- использовать `meta_cls` вместо `cls` для методов класса, потому что класс является метаклассом. Напомню, что `__new__` ведет себя как метод класса, даже без декоратора `@classmethod`.

Пример 24.18. metalib.py: `MetaKlass`

```
class MetaKlass(type):
    print('% MetaKlass body')

    @classmethod ❶
    def __prepare__(meta_cls, cls_name, bases): ❷
        args = (meta_cls, cls_name, bases)
        print(f'% MetaKlass.__prepare__{args!r}')
        return NosyDict() ❸

    def __new__(meta_cls, cls_name, bases, cls_dict): ❹
        args = (meta_cls, cls_name, bases, cls_dict)
        print(f'% MetaKlass.__new__{args!r}')
        def inner_2(self):
            print(f'% MetaKlass.__new__:inner_2({self!r})')

        cls = super().__new__(meta_cls, cls_name, bases, cls_dict.data) ❺
        cls.method_c = inner_2 ❻

        return cls ❼
```

```
    def __repr__(cls): ❽
        cls_name = cls.__name__
        return f"<class {cls_name!r} built by MetaKlass>"

print('% metalib module end')
```

❶ `__prepare__` следует объявлять как метод класса. Это не метод экземпляра, потому что когда Python вызывает `__prepare__`, конструируемого класса еще не существует.
❷ Python вызывает метод `__prepare__` метакласса, чтобы получить отображение для размещения пространства имен конструируемого класса.
❸ Вернуть экземпляр `NosyDict`, который будет использоваться в роли пространства имен.
❹ `cls_dict` – экземпляр `NosyDict`, возвращенный методом `__prepare__`.
❺ `type.__new__` требует, чтобы в последнем аргументе был настоящий словарь `dict`, поэтому я передаю ему атрибут `data` отображения `NosyDict`, унаследованного от `UserDict`.
❻ Внедрить метод во вновь созданный класс.
❼ Как обычно, `__new__` должен вернуть только что созданный объект – в данном случае новый класс.
❽ Определение `__repr__` в метаклассе позволяет настроить представление объектов класса.

До версии Python 3.6 основным применением метода `__prepare__` было предоставление объекта `OrderedDict` для хранения атрибутов конструируемого класса, так чтобы метод `__new__` метакласса мог обрабатывать эти атрибуты в том порядке, в каком они появляются в исходном коде определения пользовательского класса. Но теперь `dict` сохраняет порядок вставки, так что `__prepare__` редко бывает нужен. Его изобретательное использование будет показано в примере «Трюк с __prepare__ в метаклассе» ниже.

Результат импорта *metalib.py* на консоли Python не особенно впечатляет. Обратите внимание на знак `%` в начале строк, выведенных этим модулем:

```
>>> import metalib
% metalib module start
% MetaKlass body
% metalib module end
```

Но при импорте *evaldemo_meta.py* происходит масса интересного (см. пример 24.19).

Пример 24.19. Эксперименты с evaldemo_meta.py на консоли

```
>>> import evaldemo_meta
@ builderlib module start
@ Builder body
@ Descriptor body
@ builderlib module end
% metalib module start
% MetaKlass body
% metalib module end
# evaldemo_meta module start ❶
```

```
% MetaKlass.__prepare__(<class 'metalib.MetaKlass'>, 'Klass', ❷
                       (<class 'builderlib.Builder'>,))
% NosyDict.__setitem__(<NosyDict instance>, '__module__', 'evaldemo_meta') ❸
% NosyDict.__setitem__(<NosyDict instance>, '__qualname__', 'Klass')
# Klass body
@ Descriptor.__init__(<Descriptor instance>) ❹
% NosyDict.__setitem__(<NosyDict instance>, 'attr', <Descriptor instance>) ❺
% NosyDict.__setitem__(<NosyDict instance>, '__init__',
                       <function Klass.__init__ at …>) ❻
% NosyDict.__setitem__(<NosyDict instance>, '__repr__',
                       <function Klass.__repr__ at …>)
% NosyDict.__setitem__(<NosyDict instance>, '__classcell__', <cell at …: empty>)
% MetaKlass.__new__(<class 'metalib.MetaKlass'>, 'Klass',
                    (<class 'builderlib.Builder'>,), <NosyDict instance>) ❼
@ Descriptor.__set_name__(<Descriptor instance>,
                          <class 'Klass' built by MetaKlass>, 'attr') ❽
@ Builder.__init_subclass__(<class 'Klass' built by MetaKlass>)
@ deco(<class 'Klass' built by MetaKlass>)
# evaldemo_meta module end
```

❶ Строки, предшествующие этой, – результат импорта *builderlib.py* и *metalib.py*.
❷ Python вызывает `__prepare__`, начиная обработку предложения `class`.
❸ Перед тем как приступить к разбору тела класса, Python добавляет элементы `__module__` и `__qualname__` в пространство имен конструируемого класса.
❹ Экземпляр дескриптора создается и …
❺ … связывается с именем `attr` в пространстве имен класса.
❻ Определяются и добавляются в пространство имен методы `__init__` и `__repr__`.
❼ Закончив обработку тела класса, Python вызывает метод `MetaKlass.__new__`.
❽ Методы `__set_name__`, `__init_subclass__` и декоратор вызываются именно в таком порядке, после того как метод метакласса `__new__` вернул вновь сконструированный класс.

При запуске *evaldemo_meta.py* как скрипта вызывается `main()` и происходит еще несколько любопытных вещей (пример 24.20).

Пример 24.20. Запуск evaldemo_meta.py как программы

```
$ ./evaldemo_meta.py
[... 20 lines omitted ...]
@ deco(<class 'Klass' built by MetaKlass>) ❶
@ Builder.__init__(<Klass instance>)
# Klass.__init__(<Klass instance>)
@ SuperA.__init_subclass__:inner_0(<Klass instance>)
@ deco:inner_1(<Klass instance>)
% MetaKlass.__new__:inner_2(<Klass instance>) ❷
@ Descriptor.__set__(<Descriptor instance>, <Klass instance>, 999)
# evaldemo_meta module end
```

❶ Первые 21 строка, включая эту, такие же, как в примере 24.19.
❷ Печатается методом `obj.method_c()` в `main`; `method_c` был внедрен `MetaKlass.__new__`.

Но вернемся к идее класса `Checked` с дескрипторами `Field`, который реализует проверку типов во время выполнения, и посмотрим, как это можно сделать с помощью метакласса.

Реализация Checked с помощью метакласса

Я не хочу поощрять преждевременную оптимизацию и заумные решения, поэтому опишу воображаемый сценарий, чтобы оправдать переписывание *checkedlib.py* с добавлением атрибута `__slots__`, что требует применения метакласса. Можете пропустить его, если хотите.

Обоснование

Наш модуль *checkedlib.py* с методом `__init_subclass__` стал пользоваться популярностью в компании, поэтому в памяти производственных серверов в каждый момент времени крутятся миллионы экземпляров подклассов `Checked`.

Профилирование показало, что использование `__slots__` позволило бы уменьшить счет от облачного поставщика по двум причинам:

- снижение потребления памяти, поскольку экземплярам `Checked` будет не нужен собственный `__dict__`;
- повышение производительности за счет исключения метода `__setattr__`, который создавался только ради запрета неожиданных атрибутов, но вызывается в момент создания экземпляра и при каждом присваивании атрибуту, до того как будет вызван метод `Field.__set__`.

Следующий модуль *metaclass/checkedlib.py* является совместимой по интерфейсу заменой *initsub/checkedlib.py*. Тесты, включенные в оба модуля, идентичны, равно как и файлы *checkedlib_test.py* для *pytest*.

Сложность *checkedlib.py* скрыта от пользователя. Вот как выглядит исходный код скрипта, в котором этот пакет используется:

```
from checkedlib import Checked

class Movie(Checked):
    title: str
    year: int
    box_office: float

if __name__ == '__main__':
    movie = Movie(title='The Godfather', year=1972, box_office=137)
    print(movie)
    print(movie.title)
```

В этом коротком определении класса `Movie` используется три экземпляра проверяющего дескриптора `Field`, конфигурация `__slots__`, пять методов, унаследованных от `Checked`, и метакласс, собирающий все воедино. Единственная видимая часть `checkedlib` – базовый класс `Checked`.

Взгляните на рис. 24.4. Нотация хреновин и штуковин дополняет диаграмму классов UML, делая связи между классами и экземплярами более наглядными.

Например, класс `Movie`, в котором используется новый модуль *checkedlib.py*, является экземпляром `CheckedMeta` и подклассом `Checked`. А атрибуты `title`, `year` и `box_office` класса `Movie` – это три отдельных экземпляра `Field`. У каждого экзем-

пляра `Movie` свои собственные атрибуты `_title`, `_year` и `_box_office` для хранения значений соответствующих полей.

Теперь изучим код, начав с класса `Field` в примере 24.21.

Дескрипторный класс `Field` сейчас немного отличается. В предыдущих примерах каждый экземпляр дескриптора `Field` хранил свое значение в управляемом экземпляре, используя атрибут с тем же именем. Например, в классе `Movie` дескриптор `title` хранил значение поля в атрибуте `title` управляемого экземпляра. Поэтому от `Field` не требовалось предоставлять метод `__get__`.

Однако когда в классе, подобном `Movie`, используется `__slots__`, в нем не может быть одноименных атрибутов класса и экземпляра. Каждый экземпляр дескриптора является атрибутом класса, и теперь мы должны иметь отдельные атрибуты хранения в каждом экземпляре. В коде имена дескрипторов начинаются одним символом `_`. Поэтому у экземпляров `Field` имеются раздельные атрибуты `name` и `storage_name`, и мы реализуем метод `Field.__get__`.

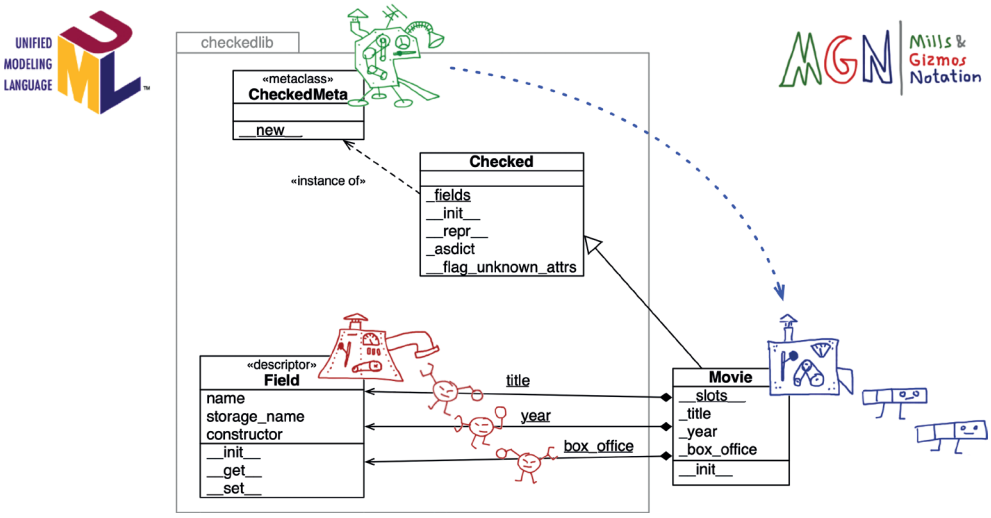


Рис. 24.4. UML-диаграмма классов, аннотирования с помощью нотации MGN: метахреновина `CheckedMeta` строит хреновину `Movie`. Хреновина `Field` строит дескрипторы `title`, `year` и `box_office`, являющиеся атрибутами класса `Movie`. Данные полей каждого экземпляра хранятся в атрибутах экземпляра `_title`, `_year` и `_box_office` класса `Movie`. Обратите внимание на границу пакета `checkedlib`. Разработчику `Movie` не нужно глубоко понимать весь механизм работы checkedlib.py

В примере 24.21 приведен исходный код `Field`, а выноски описывают только отличия от предыдущей версии.

Пример 24.21. metaclass/checkedlib.py: дескриптор `Field` с атрибутом `storage_name` и методом `__get__`

```
class Field:
    def __init__(self, name: str, constructor: Callable) -> None:
        if not callable(constructor) or constructor is type(None):
            raise TypeError(f'{name!r} type hint must be callable')
        self.name = name
```

```
        self.storage_name = '_' + name ❶
        self.constructor = constructor

    def __get__(self, instance, owner=None):
        if instance is None: ❷
            return self
        return getattr(instance, self.storage_name) ❸

    def __set__(self, instance: Any, value: Any) -> None:
        if value is ...:
            value = self.constructor()
      else:
            try:
                value = self.constructor(value)
            except (TypeError, ValueError) as e:
                type_name = self.constructor.__name__
                msg = f'{value!r} is not compatible with {self.name}:{type_name}'
                raise TypeError(msg) from e
      setattr(instance, self.storage_name, value) ❹
```

❶ Вычислить `storage_name` по аргументу `name`.
❷ Если `__get__` получает `None` в качестве аргумента `instance`, то дескриптор читается из самого управляемого класса, а не из управляемого экземпляра. Поэтому мы возвращаем дескриптор.
❸ В противном случае возвращаем значение, хранящееся в атрибуте с именем `storage_name`.
❹ Теперь `__set__` пользуется методом `setattr` для установки и изменения управляемого атрибута.

В примере 24.22 приведен код метакласса, приводящего этот пример в действие.

Пример 24.22. metaclass/checkedlib.py: метакласс `CheckedMeta`

```
class CheckedMeta(type):

    def __new__(meta_cls, cls_name, bases, cls_dict): ❶
        if '__slots__' not in cls_dict: ❷
            slots = []
            type_hints = cls_dict.get('__annotations__', {}) ❸
            for name, constructor in type_hints.items(): ❹
                field = Field(name, constructor) ❺
                cls_dict[name] = field ❻
                slots.append(field.storage_name) ❼

            cls_dict['__slots__'] = slots ❽

        return super().__new__(
            meta_cls, cls_name, bases, cls_dict) ❾
```

❶ `__new__` – единственный метод, реализованный в `CheckedMeta`.
❷ Дополнять класс, только если его `cls_dict` не содержит `__slots__`. Если `__slots__` уже присутствует, предполагаем, что это базовый класс `Checked`, а не определенный пользователем подкласс, и строим класс, ничего не добавляя.

❸ В предыдущих примерах для получения аннотаций типов мы пользовались методом `typing.get_type_hints`, но он требует передачи существующего класса в первом аргументе. В этой точке конфигурируемый класс еще не существует, поэтому нужно получать `__annotations__` непосредственно из `cls_dict` – пространства имен конструируемого класса, которое Python передает в последнем аргументе методу метакласса `__new__`.
❹ Обойти `type_hints`, чтобы ...
❺ ... построить `Field` для каждого аннотированного атрибута, ...
❻ ... перезаписать соответствующий элемент `cls_dict` экземпляром `Field` ...
❼ ... и добавить атрибут `storage_name` поля в конец списка, который нам понадобится, чтобы ...
❽ ... заполнить элемент `__slots__` в `cls_dict` – пространстве имен конструируемого класса.
❾ Напоследок вызываем `super().__new__`.

Последняя часть *metaclass/checkedlib.py* – базовый класс `Checked`, которому пользователи этой библиотеки будут наследовать, чтобы дополнить собственные классы наподобие `Movie`.

Код этой версии `Checked` отличается от кода в модуле *initsub/checkedlib.py* (см. примеры 24.5 и 24.6) в трех местах.

1. Добавлен пустой атрибут `__slots__`, который служит для метода `CheckedMeta.__new__` указанием на то, что этот класс не нуждается в специальной обработке.
2. Удален метод `__init_subclass__`. Его работу теперь выполняет метод `CheckedMeta.__new__`.
3. Удален метод `__setattr__`. Он стал не нужен, потому что добавление `__slots__` в пользовательский класс препятствует заданию необъявленных атрибутов.

В примере 24.23 приведен код окончательной версии `Checked`.

Пример 24.23. metaclass/checkedlib.py: базовый класс `Checked`

```
class Checked(metaclass=CheckedMeta):
    __slots__ = () # skip CheckedMeta.__new__ processing

    @classmethod
    def _fields(cls) -> dict[str, type]:
        return get_type_hints(cls)

    def __init__(self, **kwargs: Any) -> None:
        for name in self._fields():
            value = kwargs.pop(name, ...)
            setattr(self, name, value)
        if kwargs:
            self.__flag_unknown_attrs(*kwargs)

    def __flag_unknown_attrs(self, *names: str) -> NoReturn:
        plural = 's' if len(names) > 1 else ''
        extra = ', '.join(f'{name!r}' for name in names)
        cls_name = repr(self.__class__.__name__)
        raise AttributeError(f'{cls_name} object has no attribute{plural} {extra}')
```

```
    def _asdict(self) -> dict[str, Any]:
        return {
            name: getattr(self, name)
            for name, attr in self.__class__.__dict__.items()
                if isinstance(attr, Field)
        }

    def __repr__(self) -> str:
        kwargs = ', '.join(
            f'{key}={value!r}' for key, value in self._asdict().items()
        )
        return f'{self.__class__.__name__}({kwargs})'
```

На этом завершается третий вариант построителя классов с проверяющими дескрипторами.

В следующем разделе мы рассмотрим общие вопросы, относящиеся к метаклассам.

Метаклассы на практике

Метаклассы – штука мощная, но трудная. Прежде чем браться за реализацию метакласса, примите во внимание описанные ниже соображения.

Современные средства позволяют упростить или заменить метаклассы

Со временем появились новые языковые средства, сделавшие метаклассы ненужными в нескольких типичных ситуациях.

Декораторы классов

Проще, чем метаклассы, и меньше шансов на конфликты с базовыми классами и метаклассами.

`__set_name__`

Делает лишним привлечение метаклассов к автоматическому заданию имени дескриптора[1].

`__init_subclass__`

Предоставляет способ настройки создания класса, прозрачный для конечного пользователя и даже более простой, чем декоратор. Но при этом в сложной иерархии классов возможны конфликты.

Сохранение порядка вставки ключей во встроенном словаре

Устранило главную причину для использования `__prepare__`: предоставить `OrderedDict` для хранения пространства имен конструируемого класса. Python вызывает `__prepare__` только для метаклассов, поэтому если вам нужно было

[1] В первом издании книги в продвинутых версиях класса `LineItem` метакласс использовался только для того, чтобы задать имя для хранения атрибутов. См. код метаклассов в примерах bulkfood в репозитории для первого издания (*https://github.com/fluentpython/example-code/tree/master/21-class-metaprog/bulkfood*).

обрабатывать имена в том порядке, в каком они добавлялись в пространство имен класса в исходном коде, то до выхода версии Python 3.6 приходилось использовать метакласс.

По состоянию на 2021 год все активно поддерживаемые версии CPython поддерживали все перечисленные выше возможности.

Я горячий сторонник этих средств, потому что и так вижу в нашей профессии слишком много ненужной сложности, а метаклассы – ворота к этой сложности.

Метаклассы – стабильное языковое средство

Метаклассы были добавлены в Python 2.2 в 2002 году вместе с так называемыми «классами в новом стиле», дескрипторами и свойствами.

Удивительно, что пример `MetaBunch`, впервые опубликованный Алексом Мартелли в июле 2002 года, все еще работает в Python 3.9 – изменился только способ задания используемого метакласса: в Python 3 для этого служит синтаксическая конструкция `class Bunch(metaclass=MetaBunch):`.

Ни одно из добавлений, упомянутых в разделе «Современные средства позволяют упростить или заменить метаклассы» выше, не «ломает» существующий код с метаклассами. Но унаследованный код, в котором используются метаклассы, часто можно упростить, воспользовавшись этими средствами, особенно если вы готовы отказаться от поддержки версий Python младше 3.6 (а она уже все равно не сопровождается).

У класса может быть только один метакласс

Если в объявлении класса встречается два или более метаклассов, то выдается такое загадочное сообщение об ошибке:

```
TypeError: metaclass conflict: the metaclass of a derived class
must be a (non-strict) subclass of the metaclasses of all its bases
```

Для этого даже не нужно множественное наследование. Например, следующее объявление приведет к такой ошибке `TypeError`:

```
class Record(abc.ABC, metaclass=PersistentMeta):
    pass
```

Мы видели, что `abc.ABC` – экземпляр метакласса `abc.ABCMeta`. Если метакласс `PersistentMeta` сам не является подклассом `abc.ABCMeta`, то мы получаем конфликт метаклассов.

Бороться с этой ошибкой можно двумя способами:

- придумать, как решить задачу, обойдясь хотя бы без одного из участвующих метаклассов;
- написать собственный метакласс `PersistentABCMeta`, являющийся подклассом как `abc.ABCMeta`, так и `PersistentMeta`, и использовать его как единственный метакласс `Record`[1].

[1] Если у вас голова пошла кругом от перспективы прибегнуть к множественному наследованию метаклассов, то и хорошо. Я бы тоже держался от этого решения подальше.

Я могу себе представить, что к решению с метаклассом, наследующим двум базовым метаклассам, прибегают, чтобы не выбиться из графика. Но мой опыт показывает, что программирование метаклассов всегда занимает больше времени, чем планировалось, поэтому такой подход чреват рисками. Если вы все-таки пошли по этому пути и успели к сроку, то код может содержать тонкие ошибки. Даже если явных ошибок нет, все равно этот подход следует рассматривать как технический долг просто потому, что его трудно понять и сопровождать.

Метаклассы должны быть деталью реализации

Помимо `type`, в стандартной библиотеке Python 3.9 есть всего шесть метаклассов. Самыми известными, вероятно, являются `abc.ABCMeta`, `typing.NamedTupleMeta` и `enum.EnumMeta`. Ни один из них не предназначен для явного использования в пользовательском коде. Можно считать их деталью реализации.

Хотя метаклассы открывают возможность для эксцентричного метапрограммирования, лучше следовать принципу наименьшего удивления (*https://en.wikipedia.org/wiki/Principle_of_least_astonishment*), чтобы большинство пользователей действительно могли относиться к ним как к детали реализации[1].

В последние годы некоторые метаклассы в стандартной библиотеке Python были заменены другими механизмами, не нарушив открытый API соответствующих пакетов. Простейший способ обеспечить возможность развития таких API в будущем – предложить пользователям обычный класс, которому можно унаследовать для доступа к функциональности, предоставляемой метаклассом, как мы и поступили в приведенных примерах.

В завершение рассказа о метапрограммировании классов поделюсь самым крутым, хотя и небольшим примером метакласса, который мне удалось найти при подготовке материала к этой главе.

Метаклассный трюк с __prepare__

Перерабатывая эту главу для второго издания, я хотел найти простые, но убедительные примеры для замены класса `LineItem` из модуля *bulkfood*, в котором после выхода Python 3.6 необходимость в метаклассах отпала.

Самую простую и вместе с тем самую интересную идею применения метакласса подал мне Жоао С. О. Буэно, которого в бразильском сообществе Python лучше знают под ником JS. Одно из применений этой идеи – создание класса, который автоматически генерирует числовые константы:

```
>>> class Flavor(AutoConst):
...     banana
...     coconut
...     vanilla
...
>>> Flavor.vanilla
2
```

[1] Я зарабатывал кодированием с применением Django в течение нескольких лет, прежде чем решил поинтересоваться, как в нем реализованы поля модели. И только тогда я узнал о дескрипторах и метаклассах.

```
>>> Flavor.banana, Flavor.coconut
(0, 1)
```

Да, этот код именно так и работает! На самом деле это тест, включенный в файл *autoconst_demo.py*.

Ниже приведен ориентированный на пользователей базовый класс `AutoConst` и стоящий за ним метакласс, реализованные в файле *autoconst.py*:

```
class AutoConstMeta(type):
    def __prepare__(name, bases, **kwargs):
        return WilyDict()

class AutoConst(metaclass=AutoConstMeta):
    pass
```

И это всё.

Понятно, что весь фокус кроется в `WilyDict`.

Когда Python обрабатывает пространство имен пользовательского класса и читает `banana`, он ищет это имя в отображении, предоставляемом методом `__prepare__`: экземпляре класса `WilyDict`. `WilyDict` реализует метод `__missing__`, описанный в разделе «Метод __missing__» главы 3. Изначально в экземпляре `WilyDict` нет ключа `'banana'`, поэтому вызывается метод `__missing__`. Он динамически создает запись с ключом `'banana'` и значением `0` и возвращает это значение. Python вполне удовлетворен и переходит к поиску ключа `'coconut'`. `WilyDict` с готовностью добавляет запись с таким ключом и значением `1` и возвращает значение. То же самое происходит с ключом `'vanilla'`, который отображается на `2`.

Мы уже встречали `__prepare__` и `__missing__` раньше. Новшество в том, как JS соединил их вместе.

Ниже приведен исходный код `WilyDict`, также находящийся в файле *autoconst.py*:

```
class WilyDict(dict):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__next_value = 0

    def __missing__(self, key):
        if key.startswith('__') and key.endswith('__'):
            raise KeyError(key)
        self[key] = value = self.__next_value
        self.__next_value += 1
        return value
```

Экспериментируя, я обнаружил, что Python ищет `__name__` в пространстве имен конструируемого класса, что заставляет `WilyDict` добавить запись `__name__` и увеличить `__next_value`. Поэтому я добавил в `__missing__` предложение `if`, которое возбуждает исключение `KeyError` для ключей, выглядящих как dunder-атрибуты.

Пакет *autoconst.py* требует уверенного владения механизмом построения динамических классов в Python и вместе с тем иллюстрирует его.

Я отлично провел время, расширяя функциональность `AutoConstMeta` и `AutoConst`, но вместо того чтобы делиться своими экспериментами, оставлю вам шанс получить удовольствие, развивая этот изобретательный трюк JS.

Вот несколько идей:

- добавьте возможность получить имя константы по ее значению. Например, выражение `Flavor[2]` должно возвращать `'vanilla'`. Это можно сделать, реализовав метод `__getitem__` в `AutoConstMeta`. Начиная с Python 3.9 можно реализовать метод `__class_getitem__` в самом `AutoConst`;
- добавьте поддержку итерирования по классу, реализовав метод `__iter__` в метаклассе. Я бы заставил `__iter__` отдавать константы в виде пар `(name, value)`;
- реализуйте новый вариант `Enum`. Это будет непростым предприятием, потому что пакет `enum` полон разных трюков, в частности в нем есть метакласс `EnumMeta`, содержащий сотни строк кода и нетривиальный метод `__prepare__`.

Дерзайте!

Специальный метод `__class_getitem__` был добавлен в версию Python 3.9 для поддержки обобщенных типов как часть документа PEP 585 «Type Hinting Generics In Standard Collections» (*https://peps.python.org/pep-0585/*). Благодаря `__class_getitem__` разработчики ядра Python избавились от необходимости писать новый метакласс для реализации `__getitem__` во встроенных типах, чтобы мы могли писать аннотации обобщенных типов вида `list[int]`. Это узкая сфера применения, но она дает пример потенциального использования метаклассов: реализация операторов и других специальных методов, работающих на уровне класса, например итерирование по самому классу, как в подклассах `Enum`.

Заключение

Метаклассы, равно как декораторы классов и метод `__init_subclass__`, полезны для решения следующих задач:

- регистрация подкласса;
- структурная проверка подкласса;
- применение декораторов сразу к нескольким методам;
- сериализация объектов;
- объектно-реляционное отображение;
- постоянное хранение объектов;
- реализация специальных методов на уровне класса;
- реализация средств работы с классами, имеющихся в других языках, например характеристик (*https://en.wikipedia.org/wiki/Trait_(computer_programming)*) и аспектно-ориентированного программирования (*https://en.wikipedia.org/wiki/Aspect-oriented_programming*).

Метапрограммирование классов иногда помогает решить проблемы производительности, позволяя выполнять на этапе импорта действия, которые иначе пришлось бы многократно повторять на этапе выполнения.

Подводя итоги, хочу напомнить заключительный совет Алекса Мартелли из эссе «Водоплавающие птицы и ABC», приведенного в главе 13.

> И *не* определяйте свои ABC (или метаклассы) в производственном коде. Если вам кажется, что без этого не обойтись, держу пари, что это, скорее всего, желание поскорее забить гвоздь, раз уж в руках молоток, – вам (и тем, кому предстоит сопровождать вашу программу) будет куда комфортнее иметь дело с прямолинейным и простым кодом, где нет таких глубин.

Я думаю, что совет Мартелли применим не только к ABC и метаклассам, но также к иерархиям классов, перегрузке операторов, декораторам функциям, дескрипторам, декораторам классов и построителям классов, в которых используется метод `__init_subclass__`.

Эти мощные инструменты предназначены в первую очередь для разработки библиотек и каркасов. Естественно, приложения могут *использовать* эти инструменты, поскольку они предоставляются стандартной библиотекой и внешними пакетами. Но решение *реализовать* их в коде приложения часто является признаком преждевременного абстрагирования.

> Хорошие каркасы извлекаются, а не изобретаются[1].
>
> – Дэвид Хейнемейер Ханссон, создатель Ruby on Rails

Резюме

Мы начали эту главу с обзора атрибутов, встречающихся в объектах классов, в т. ч. `__qualname__`, и метода `__subclasses__()`. Затем мы видели, как встроенный класс `type` можно использовать для конструирования классов во время выполнения.

Мы познакомились со специальным методом `__init_subclass__` в процессе первой попытки создать базовый класс `Checked`, предназначенный для замены аннотаций типа атрибутов в пользовательских подклассах экземплярами `Field`, которые применяют конструкторы для проверки типа этих атрибутов на этапе выполнения.

Та же идея была реализована в декораторе класса `@checked`, который добавляет новые возможности в пользовательский класс по аналогии с тем, что делает метод `__init_subclass__`. Мы также видели, что ни `__init_subclass__`, ни декоратор класса не могут динамически сконфигурировать `__slots__`, потому что работают уже после создания класса.

С помощью экспериментов мы прояснили понятия «этап импорта» и «этап выполнения», продемонстрировав, в каком порядке выполняется Python-код, в котором присутствуют модули, дескрипторы, декораторы классов и метод `__init_subclass__`.

Рассмотрение метаклассов началось с объяснения того, что `type` является метаклассом, и того, как определенные пользователем метаклассы могут реализовать метод `__new__` для настройки создаваемых классов. Затем мы представили первый пользовательский метакласс, классический прием `MetaBunch` с использованием `__slots__`. Наш следующий эксперимент показал, что методы `__prepare__` и `__new__` метакласса вызываются раньше, чем `__init_subclass__` и декораторы классов, что открывает возможности для более глубокой настройки классов.

Далее был представлен третий подход к построителю классов `Checked` с дескрипторами `Field` и пользовательской конфигурацией `__slots__`, после чего мы поделились общими соображениями о применении метаклассов на практике.

---

[1] Эта фраза разошлась на цитаты. Я нашел одно из ранних прямых упоминаний в сообщении в блоге DHH, датированном 2005 годом.

Наконец, мы продемонстрировали класс `AutoConst`, придуманный Жоао С. О. Буэно и основанный на хитроумной идее метакласса с методом `__prepare__`, который возвращает отображение, реализующее метод `__missing__`. Насчитывающий меньше 20 строк код, скрипт *autoconst.py* показывает, как можно с пользой комбинировать приемы метапрограммирования в Python.

Я еще не встречал языка, которому удалось так удачно совместить простоту для начинающих, практичность для профессионалов и увлекательность для хакеров, как это сделано в Python. Спасибо Гвидо ван Россуму и всем, кто внес свой вклад в достижение этой цели.

Дополнительная литература

Калеб Хэттинг, технический рецензент этой книги, написал пакет *autoslot*, который предоставляет метакласс для автоматического создания атрибута `__slots__` в пользовательском классе. Для этого он исследует байт-код метода `__init__` и находит все присваивания атрибутам `self`. Метакласс полезен сам по себе, а кроме того, является прекрасным учебным примером: всего 74 строки в файле *autoslot.py*, включая 20 строк комментариев, в которых объясняются наиболее трудные места.

Основными справочными материалами к этой главе являются раздел 3.3.3 «Настройка создания класса» (*https://docs.python.org/3/reference/datamodel.html#customizing-class-creation*) главы «Модель данных» справочного руководства по языку Python, в котором рассматриваются метод `__init_subclass__` и метаклассы. Документация по классу `type` (*https://docs.python.org/3/library/functions.html#type*) в разделе «Встроенные функции» и раздел 4.13 «Специальные атрибуты» (*https://docs.python.org/3/library/stdtypes.html#special-attributes*) главы «Встроенные типы» руководства по стандартной библиотеке также важны.

В документации по модулю `types` из стандартной библиотеки (*https://docs.python.org/3/library/types.html*) рассматриваются две новые функции в Python 3.3, призванные оказать помощь в метапрограммировании классов: `types.new_class` и `types.prepare_class`.

Декораторы классов были формально определены в документе PEP 3129 «Class Decorators» (*https://peps.python.org/pep-3129/*), который написал Коллин Уинтер, а эталонную реализацию предложил Джек Дидерих. Доклад Джека Дидериха на конференции PyCon 2009 «Class Decorators: Radically Simple» (видео – по адресу *https://www.youtube.com/watch?v=cAGliEJV9_o*) содержит краткое введение в эту функциональность. Помимо `@dataclass`, интересным – и гораздо более простым – примером декоратора класса в стандартной библиотеке Python является `functools.total_ordering` (*https://docs.python.org/3/library/functools.html#functools.total_ordering*), который генерирует специальные методы для сравнения объектов.

Основным справочником по метаклассам в документации Python является документ PEP 3115 «Metaclasses in Python 3000» (*https://peps.python.org/pep-3115/*), в котором был предложен специальный метод `__prepare__`.

Книга Alex Martelli, Anna Ravenscroft, Steve Holden «Python in a Nutshell», 3-е издание, хотя и авторитетна, но написана раньше, чем вышел документ PEP 487 «Simpler customization of class creation» (*https://peps.python.org/pep-0487/*).

Главный пример метакласса в этой книге, MetaBunch, по-прежнему актуален, потому что его нельзя написать с помощью более простых механизмов. В книге Brett Slatkin «Effective Python», 2-е издание (Addison-Wesley), есть несколько актуальных примеров на тему построения классов, в т. ч. с использованием метаклассов.

Тем, кто хочет узнать об истоках метапрограммирования классов в Python, я рекомендую статью «Unifying types and classes in Python 2.2» (*https://www.python.org/download/releases/2.2.3/descrintro/*), написанную Гвидо ван Россумом в 2003 году. Она относится и к современному Python, поскольку охватывает то, что тогда называлось семантикой классов «нового стиля» и стало семантикой по умолчанию в Python 3, в т. ч. дескрипторы и метаклассы. Гвидо ссылается, в частности, на книгу Ira R. Forman, Scott H. Danforth «Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming» (Addison-Wesley), которой он поставил 5 звезд на сайте *Amazon.com*, сопроводив таким комментарием:

> **Эта книга внесла вклад в дизайн метаклассов в Python 2.2**
>
> Жаль, что она больше не переиздается; я продолжаю считать ее лучшей из известных мне работ на трудную тему кооперативного множественного наследования, поддерживаемого в Python с помощью функции `super()`[1].

Если вы увлеклись метапрограммированием, то, возможно, хотели бы, чтобы в Python был реализован механизм, являющийся королем всех средств метапрограммирования: синтаксические макросы, имеющиеся в семействе языков Lisp, а из более современных – в Elixir и Rust. Синтаксические макросы – механизм более мощный и менее подверженный ошибкам, чем примитивные подстановочные макросы в языке C. Это специальные функции, которые преобразуют исходный код, написанный с применением пользовательского синтаксиса, в стандартный код до этапа компиляции, что позволяет разработчикам внедрять новые языковые конструкции, не изменяя компилятор. Как и перегрузку операторов, синтаксические макросы можно употребить во зло. Но при условии, что сообщество понимает потенциальные недостатки и умеет их купировать, они способны поддержать мощные и дружественные по отношению к пользователю абстракции, в частности DSL (предметно-ориентированные языки). В сентябре 2020 года разработчик ядра Python Марк Шеннон опубликовал документ PEP 638 «Syntactic Macros» (*https://peps.python.org/pep-3129/*), в котором предложил именно это. Через год после публикации PEP 638 все еще имел статус черновой редакции и его активного обсуждения не было. Очевидно, что для разработчиков ядра Python он не стоит на первом месте. Я хотел бы, чтобы обсуждение PEP 638 возобновилось и в конечном итоге увенчалось одобрением. Синтаксические макросы позволили бы сообществу Python экспериментировать с вызывающими споры новыми средствами, например оператором `:=` (PEP 572), сопоставлением с образцом (PEP 634) и альтернативными правилами вычисления аннотаций типов (PEP 563 и 649), прежде чем вносить изменения в ядро языка. А пока можете получить представление о синтаксических макросах из пакета MacroPy (*https://github.com/lihaoyi/macropy*).

[1] Я купил подержанный экземпляр, это оказалось весьма трудное чтение.

Поговорим

Эту последнюю врезку «Поговорим» я начну длинной цитатой из Брайана Харви (Brian Harvey) и Мэттью Райта (Matthew Wright), двух профессоров информатики из Калифорнийского университета (в Беркли и Санта-Барбаре). В книге «Simply Scheme: Introducing Computer Science» (MIT Press) Харви и Райт пишут:

Есть два направления в преподавании информатики. Схематично их можно представить следующим образом:

1. **Консервативный взгляд.** Компьютерные программы стали настолько большими и сложными, что человеческий мозг не в состоянии их охватить. Поэтому задача обучения информатике заключается в том, чтобы научить людей дисциплинированной работе, когда 500 средних программистов, собравшись вместе, могут написать программу, отвечающую спецификации.
2. **Радикальный взгляд.** Компьютерные программы стали настолько большими и сложными, что человеческий мозг не в состоянии их охватить. Поэтому задача обучения информатике заключается в том, чтобы научить людей расширять сознание, чтобы в нем хватило места всей программе. Для этого нужно учить мыслить более крупными, более эффективными, более гибкими категориями, не ограничиваясь очевидными вещами. Каждая единица программистской мысли должна давать большую отдачу в терминах возможностей программы[1].

– Брайан Харви и Мэттью Райт, предисловие к книге «Simply Scheme»[2]

Карикатурные описания Харви и Райта относятся к преподаванию информатики, но они применимы и к проектированию языков программирования. Вы, наверное, уже догадались, что я приверженец «радикальной» точки зрения и полагаю, что Python проектировался именно в таком ключе.

Идея свойств – большой шаг вперед по сравнению с подходом Java, требующим применять акцессоры с самого начала. Этот подход поддерживается всеми Java IDE с помощью комбинации клавиш для генерации методов чтения и установки. Основное достоинство свойств заключается в том, что они позволяют начать разработку класса, сделав атрибуты открытыми – в духе принципа *KISS*, – понимая при этом, что в любой момент открытый атрибут можно, не прилагая особых усилий, сделать свойством. Но дескрипторы идут еще дальше, они предоставляют механизм для абстрагирования повторяющейся логики акцессоров. Этот механизм настолько эффективен, что используется и в некоторых конструкциях самого языка Python.

Еще одна плодотворная идея – функции как полноправные объекты. Она прокладывает дорогу к функциям высшего порядка. Как выясняется, комбинация дескрипторов и функций высшего порядка позволяет унифицировать функции и методы. Метод `__get__` функции порождает объект метода на лету путем привязки экземпляра к аргументу `self`. Это элегантное решение[3].

---

[1] *Brian Harvey, Matthew Wright.* Simply Scheme. MIT Press, 1999. С. xvii. Полный текст имеется на сайте Berkeley.edu (*https://www.eecs.berkeley.edu/~bh/ss-toc2.html*).

[2] См. стр. xvii. Полный текст доступен на сайте Berkeley.edu (*https://people.eecs.berkeley.edu/~bh/ss-toc2.html*).

[3] Книга «Machine Beauty» Дэвида Гелернтера (Basic Books) открывается интригующим обсуждением элегантности и эстетики в работе инженера: от мостов до программного обеспечения. Последующие главы не так хороши, но начало оправдывает потраченные деньги.

Наконец, полноправными объектами являются и классы. Нельзя не восхититься дизайном, при котором доступный начинающим программистам язык предоставляет столь мощные концепции, как декораторы классов и полноценные пользовательские метаклассы. И что поразительно: продвинутые средства интегрированы в язык таким образом, что не усложняют его применение для эпизодического программирования (а даже помогают – под капотом). Своим удобством и успешностью такие каркасы, как Django и SQLAlchemy, во многом обязаны метаклассам. Со временем метапрограммирование классов в Python становится все проще и проще, по крайней мере в типичных ситуациях. Лучшие языковые средства – те, от которых выигрывают все, даже если некоторые пользователи Python о них и не знают. Но они всегда могут поучиться и написать свою, не менее грандиозную библиотеку.

Я с нетерпением буду ждать вашего вклада в жизнь сообщества и в экосистему Python!

Послесловие

> Python – язык, разрешающий взрослым все.
>
> – Алан Раньян, сооснователь Plone

Афористичное определение Алана подчеркивает одно из лучших качеств Python: он отходит в сторону и дает возможность делать то, что вам необходимо. Это также означает, что он не предоставляет средств, с помощью которых вы могли бы наложить ограничения на то, что другие могут делать с вашим кодом и создаваемыми в нем объектами.

В возрасте 30 лет Python все еще набирает популярность. Конечно, Python не идеален. Лично меня подчас раздражает разнобой в именовании идентификаторов в стандартной библиотеке, например `CamelCase`, `snake_case` и `joinedwords`. Но определение языка и стандартная библиотека – лишь часть экосистемы. А лучшую ее часть составляет сообщество пользователей и авторов.

Приведу пример высоких качеств сообщества. Как-то утром писал я о `asyncio` и впал в отчаяние из-за того, что в API так много функций, из которых десятки являются сопрограммами, а сопрограммы нужно вызывать с помощью `yield from`, тогда как к обычным функциям эта конструкция неприменима. Все это описано в документации по `asyncio`, но иногда приходится прочитать несколько абзацев, пока до тебя дойдет, что некоторая функция является сопрограммой. Поэтому я отправил сообщение в список рассылки python-tulip, назвав его «Предложение: выделять сопрограммы в документации по asyncio» (*https://groups.google.com/forum/#!topic/ python-tulip/Y4bhLNbKs74*). К беседе подключились Виктор Стиннер, разработчик ядра `asyncio`, Андрей Светлов, основной автор `aiohttp`, Бен Дарнелл, главный разработчик Tornado, и Глиф Лефковиц, автор Twisted. Дарнелл предложил решение, Александр Шорин объяснил, как реализовать его в Sphinx, а Стиннер внес необходимые изменения в конфигурацию и разметку. Менее чем через 12 часов после моего вопроса вся выложенная в сеть документация по `asyncio` была обновлена – в ней появились метки *coroutine* (*https://docs.python.org/3/library/asyncio-eventloop.html#executor*).

Это произошло не в каком-то клубе для избранных. Любой может войти в список python-tulip, и я сам, внося это предложение, отправил несколько сообщений. Эта история лишний раз демонстрирует, что сообщество действительно открыто для новых идей и новых членов. Гвидо ван Россум постоянно присутствует в python-tulip и регулярно отвечает даже на простые вопросы.

Приведу еще один пример открытости: задачей фонда Python Software Foundation (PSF) является увеличение разнообразия в сообществе Python. Уже получены обнадеживающие результаты. В правление фонда в период 2013–2014 годов впервые были избраны женщины: Джессика Маккеллар и Линн Рут. А на конференции 2015 PyCon North America в Мореале – под председательством Дианы Кларк – примерно треть выступавших были женщинами. PyLadies

стало поистине глобальным движением, и я горжусь тем, что у нас в Бразилии так много отделений PyLadies.

Если вы пишете на Python, но еще не присоединились к сообществу, призываю вас не откладывать с этим. Ищите отделение PyLadies или группу пользователей Python (Python Users Group –PUG) в своем регионе. Если такой еще нет, создайте ее сами. Python присутствует везде, поэтому вы не будете одиноки. Посещайте мероприятия, если есть такая возможность. В том числе онлайновые. Во время пандемии Covid-19 я много узнал на «встречах в холле» онлайновых конференций. Приезжайте на конференцию PythonBrasil – мы уже много лет предоставляем слово докладчикам из других стран. Личные встречи с коллегами-программистами дают куда больше, чем общение в сети; известны примеры, когда они приносили реальные плоды, выходящие за рамки обмена знаниями. Как работа и дружба в «реале».

Я точно не мог бы написать эту книгу без помощи многочисленных друзей, которыми за годы работы обзавелся в сообществе Python.

Мой отец, Хайро Рамальо, частенько говаривал «Só erra quem trabalha» – «не ошибается тот, кто ничего не делает» по-португальски. Это прекрасный совет тем, кого парализует страх наделать ошибок. Уж я-то наделал их массу, когда писал эту книгу. Рецензенты, редакторы и читатели предварительной версии отловили многие из них. Не прошло и нескольких часов с момента публикации первой предварительной версии, как один читатель сообщил об опечатках на странице ошибок. Этот читатель был отнюдь не единственным, а друзья обращались ко мне напрямую со своими советами и поправками. Корректоры из издательства O'Reilly найдут и другие ошибки на этапе производства книги, который начнется сразу после того, как я наконец закончу ее писать. Я принимаю на себя ответственность и приношу извинения за те ошибки, которые останутся, а также за стилистические погрешности.

Я рад, что работа все же подошла к концу, несмотря на все ошибки и трудности, и в высшей степени благодарен всем, что помогал мне на этом пути.

Надеюсь вскоре встретиться с вами на каком-нибудь реальном мероприятии. Не стесняйтесь поздороваться, если наткнетесь на меня!

Дополнительная литература

В конце книги я хочу привести ссылки на ресурсы, в которых объясняется, что такое «дух Python» – основной вопрос, на который я пытался ответить в этой книге.

Брэндон Родес – блестящий преподаватель Python, а его доклад «A Python Æsthetic: Beauty and Why I Python» (*https://www.youtube.com/watch?v=x-kB2o8sd5c*) великолепен, начиная уже с использования символа Unicode U+00C6 (LATIN CAPITAL LETTER AE) в названии. Другой замечательный преподаватель, Раймонд Хэттингер, говорил о красоте в Python на конференции PyCon US в своем выступлении «Transforming Code into Beautiful, Idiomatic Python» (*https://www.youtube.com/watch?v=OSGv2VnC0go*).

Стоит почитать обсуждение «The Evolution of Style Guides» (*https://mail.python.org/pipermail/python-ideas/2015-March/032557.html*), начатое Яном Ли в списке рассылки Python-ideas. Ли отвечает за сопровождение пакета `pep8` (*https://pypi.python.org/pypi/pep8/*), который проверяет исходный Python-код на совмес-

тимость с документом PEP 8. Для проверки кода в этой книге я пользовался программами `flake8` (*https://pypi.python.org/pypi/flake8*), которая обертывает `pep8`, `pyflakes` (*https://pypi.python.org/pypi/pyflakes*) и плагином McCabe для оценки сложности, написанным Нэдом Бэтчелдером (*https://pypi.python.org/pypi/mccabe*).

Помимо PEP 8, есть и другие авторитетные стилистические руководства: Google Python Style Guide (*https://google-styleguide.googlecode.com/svn/trunk/pyguide.html*) и Pocoo Style Guide (*http://www.pocoo.org/internal/styleguide/*), предложенное командой, подарившей нам Flake, Sphinx, Jinja 2 и другие не менее замечательные библиотеки на Python.

Руководство автостопщика по Python (Hitchhiker's Guide to Python!) (*http://docs.python-guide.org/en/latest/*) – коллективный труд, посвященный написанию кода в духе Python. Наибольший вклад в него внес Кеннет Рейц, легендарный герой сообщества, прославившийся своим образцово «питоническим» пакетом `requests`. Дэвид Гуджер представил на конференции PyCon US 2008 пособие под названием «Code Like a Pythonista: Idiomatic Python» (*https://david.goodger.org/projects/pycon/2007/idiomatic/handout.html*). В печатном виде оно занимает 30 страниц. Гуджер создал `reStructuredText` и `docutils`, положенные в основу Sphinx, великолепной системы документирования для Python (кстати говоря, в ней подготовлена и официальная документация по MongoDB и многим другим проектам).

Мартин Фаассен поднимает вопрос «Что такое дух Python?» в своем блоге (*http://blog.startifact.com/posts/older/what-is-pythonic.html*) В списке рассылки `python-list` есть обсуждение с таким же заголовком (*http://bit.ly/1e8raAA*). Статья Мартина относится к 2005 году, а это обсуждение – к 2003-му, но «питонический» идеал изменился не сильно – как, впрочем, и сам язык. Из обсуждений, в заголовке которых встречается слово «Pythonic», хотелось бы выделить «Pythonic way to sum n-th list element?» (*https://mail.python.org/pipermail/python-list/2003-April/192027.html*), которое я обильно цитировал во врезке «Поговорим» в главе 12.

В документе PEP 3099 «Things that will Not Change in Python 3000» (*https://www.python.org/dev/peps/pep-3099/*) объясняется, почему многие вещи реализованы так, а не иначе, даже после масштабной переработки Python при выходе версии 3. Долгое время Python 3 называли Python 3000, но он появился на несколько столетий раньше – приведя некоторых в смятение. Автор документа PEP 3099, Георг Брандл, собрал многие высказывания нашего пожизненного великодушного диктатора, Гвидо ван Россума. На странице Python Essays (https://www.python.org/doc/essays/) опубликовано несколько текстов самого Гвидо.

Предметный указатель

@asyncio.coroutine, декоратор 721
@ знак 540
@cached_property, декоратор 790
@contextmanager, декоратор 617
@dataclass
  аргументы по умолчанию 190
  именованные аргументы 190
  переменные только для инициализации 196
  метод __hash__ 191
  пример использования 197
  постинициализация 194
  опции полей 191
  типизированные атрибуты класса 196
@typing.overload, декоратор 492
^, оператор 393
+=, оператор 77, 545
+ELLIPSIS, директива 35
+ оператор 530
+, оператор 38, 76, 530, 533
+x 531
<=, оператор 542
<, оператор 542
== оператор 213, 230, 393, 542
:= оператор 53
!=, оператор 542
*=, оператор 77, 545
>=, оператор 542
>, оператор 542
|=, оператор 102
|, оператор 102
~, оператор 530
404, код ошибки (Не найдено) 711
% (деление по модулю), оператор 33
** (двойная звездочка) 242
** (двойная звездочка), оператор 101
__ (двойное подчеркивание) 32
__abs__ 38
__add__ 38, 78, 533, 548
.add_done_callback(), метод 698
.append(), метод 90
*args 62
__bool__ 41
__builtins__ 99
__bytes__ 353
__call__ 240
__class__ 802, 835
__contains__ 35, 114
__del__ 225
__delattr__ 804
__delete__ 810
__dict__ 99, 802
__dir__ 804
__doc__ 235
__enter__ 613
__eq__ 393
.__eq__ 542
__exit__ 613
__float__ 363
__format__ 353, 358, 399, 408
.frombytes(), метод 356
__get__ 810
__getattr__ 390, 804
__getattribute__ 804
__getitem__ 33, 111, 388
__hash__ 363, 393
__iadd__ 78, 549
_ (подчеркивание) 67, 369
-, оператор 530
__init__ 37, 571, 779
__init_subclass__ 840
__int__ 363
__invert__ 530
.items(), метод 129
__iter__ 559, 562
.keys() метод 129
__len__ 33, 45, 388
__missing__ 112
__mro__ 440, 835
__mul__ 38, 538
__name__ 835
__neg__ 530
__new__ 779, 808

__next__ 562, 566
__pos__ 530
__post_init__ 194
__prepare__ 871
__radd__ 535
__repr__ 38, 40, 353
__rmul__ 538
__set__ 810, 815
__setattr__ 805
__setitem__ 74, 418
__set_name__ 840
__slots__ 370, 393, 802, 846
__str__ 40, 353
__subclasshook__ 441
[], квадратные скобки 35, 52, 74
коллекций API 41
\\ (обратная косая черта) 52
[:], оператор 216
*, оператор 38, 76, 242, 538
#, оператор 45
(), оператор вызова 240
... (многоточие) 75, 383
(), скобки 52
*_, символ 68
\\N{} (нотация литералов Unicode) 152
{}, фигурные скобки 52
.pop(), метод 90
.result(), метод 698
!r, поле преобразования 40
%r, спецификатор 40

A

ABC (абстрактный базовый класс)
  виртуальные подклассы 438
  аннотации типов 277
  в стандартной библиотеке 427
  гусиная типизация 422
  использование функции register 440
  определение и использование 430
  синтаксические детали 435
  создание подкласса 426
  создание подкласса гусиная типизация 426
  создание подклассов ABC 435
  структурная типизация 440
  UML-диаграмма классов 42
abc.ABC, класс 428
abc.Iterable 280
abc.Sequence 280
and, оператор 530
Any, тип 266
array.array, класс 356
asciize, функция 161
asyncio, пакет
  документация 720
  достижение максимальной производительности 728
  загрузка файлов 725
  реализация очереди 93
  пример скрипта 721
  улучшение асинхронного загрузчика 730
asyncio, пакет
  предоставляемые API 93
asyncpg 729

B

bisect, модуль 82
black, инструмент 260
blue, инструмент 260
BOM (маркер порядка байтов) 146
bool, тип 41
bool(x) 41
bytearray, тип 137

C

Callable, абстрактный базовый класс 429
Callable, тип 288
callable(), функция 240
chain, генератор 591
ChainMap 116
Chardet, библиотека 146
classmethod, декоратор 357
clock, декоратор
  параметризованный 324
  на основе класса 327
cls.__bases 835
cls.mro() 836
cls.__qualname__ 835
cls.__subclasses__() 835
collections.abc, модуль
  включенные ABC 428
  ChainMap 116
  collections.deque 90
  collections.MutableSequence 127, 426
  Counter 117
  defaultdict 111
  defaultdict и OrderedDict 106
  множественное наследование 475

Mapping и MutableMapping 105, 429
UserDict 118
collections.deque класс 90
collections.namedtuple, класс 34, 181
concurrent.futures
загрузка файлов 696
запуск процессов 701
Container, абстрактный базовый класс 428
Container, интерфейс 42
copy, функция 218
Counter 117
CPython 225
Curio, проект 760

D

Dask 675
decimal.Decimal, класс 531
deepcopy, функция 218
defaultdict 106, 111
default_factory 111
del, предложение 75, 800
deque (двусторонняя очередь) 83, 90
dis, модуль 306
Django, каркас 477
doctest, пакет для тестирования
+ELLIPSIS, директива 35
dunder-методы 33

E

EAFP, принцип 639
else, блоки 638
Executor.map 704

F

FastAPI, каркас 742
FIFO (первым пришел, первым ушел) 83
flags2_common.py, модуль 707
flake8, инструмент 260
f-строки
делегирование форматирования 358
преимущества 33
специальные методы для строкового форматирования 40
fold_equal, функция 158
for/else, комбинация 638
ForkingMixin 476
for, циклы 398
fromhex, метод класса 139
frozenset 123
fsdecode(), функция 170
fsencode(), функция 170
functools, модуль
декоратор cache 315
кеширование свойств 789
functools.lru_cache 315
functools.reduce 394, 588
functools.singledispatch 318
functools.wraps, декоратор 314
фиксация аргументов 248
futures.as_completed 713

G

gevent, библиотека 660
greenlet, пакет 660

H

Hashable, абстрактный базовый класс 429
heapq, пакет 93
HTTPServer, класс 475
HTTPX, библиотека 725

I

is, оператор 213, 530
isinstance, функция 425
Iterable, абстрактный базовый класс 428
Iterable, интерфейс 42
iter, функция
и специальный метод __iter__ 37
перебор слов 557
itertools, модуль 577

J

JSON-подобные данные 775
Jupyter, проект 675

K

key, аргумент 98
KISS, принцип 408, 490, 687

L

LBYL, принцип 639
Least Recently Used (LRU) 317
lis.py, интерпретатор
класс Environment 626
класс Procedure 636
предложения импорта и типы 623
синтаксический анализатор 624
синтаксис Scheme 622
сопоставление с образцом 69

цикл REPL 628
функция evaluate 629
OR-образцы 637
list.sort, метод 81
locale.strxfrm, функция 163
LRU. См. Least Recently Used
lru_cache, декоратор 315

M

map, функция 396
Mapping, абстрактный базовый класс 429
MappingProxyType, класс-обертка 120
MappingView, абстрактный базовый класс 429
match/case, предложение 65, 102
memoryview, класс 86
MGN (нотация хреновин и штуковин) 813
MRO (порядок разрешения методов) 440, 473
multiprocessing, пакет 93, 655
MutableMapping, абстрактный базовый класс 105
MutableSet 127
Mypy, программа проверки типов 256

N

namedtuple 181
nfc_equal, функция 158
NFKC/NFKD, нормализация 156
nonlocal, объявление 300
NoReturn, тип 291
not, оператор 530
NotImplemented 536
NotImplementedError 536
numbers, пакет 278, 453
абстрактные базовые классы 453
NumPy 88

O

operator, модуль 245
Optional, тип 269
OrderedDict 106, 115
OR-образцы 637
or, оператор 530
os, модуль 170
os, модуль, использование типов str и bytes 170
os.walk, генераторная функция 578

P

pickle, модуль 117
Pingo, библиотека 120
property, класс 794
property, функция 314
PSF. См. Python Software Foundation
PUG. См. Python, группы пользователей
PyLadies 878
Python
работа на многоядерных процессорах 673
сайт fluentpython.com 25
сообщество 878
функциональное программирование 251
Python, группы пользователей 879
Python Software Foundation 878
python-tulip, список рассылки 878
PyTorch 675
pyuca, библиотека 164, 171
PyUCA, модуль 164

Q

queue, пакет 93

R

reduce, функция 393, 395, 588
re.findall, функция 571
re.finditer, функция 571
register, метод 438, 440
reprlib, модуль 383
requests, библиотека 695

S

sanitize.py, модуль 160
S-выражение 622
Scheme, язык 69, 622
SciPy 88
self, аргумент 815, 832
Sentence, классы 564
Sequence, абстрактный базовый класс 429
Set, абстрактный базовый класс 429
setlocale, функция 163
shelve, модуль 117
singledispatch, декоратор 318
Sized, абстрактный базовый класс 428
Sized, интерфейс 42
staticmethod, декоратор 357
str.casefold, метод 158

str.format, метод 40, 358
str.format(), метод 33
StrKeyDict 118
str, функция 37
struct, модуль 136
SyntaxError, исключение 144

T

TCP-сервер 746
TensorFlow 675
ThreadingHTTPServer, класс 476
ThreadingMixIn, класс 476
t[:], оператор 226
Timsort,
  алгоритм сортировки 98
Tkinter, инструмент разработки GUI
  достоинства и недостатки 485
  множественное наследование 480
TQDM, пакет 707, 713
try/else, комбинация 638
Twisted, библиотека 770
TypedDict 176, 498
Typeshed, проект 279
TypeVar 280
typing, модуль 266
typing.NamedTuple 184

U

UML-диаграммы классов
  ABC в collections.abc 428
  для класса TkInter Widget и его суперклассов 472
  для паттерна проектирования Команда 345
  для паттерна проектирования Стратегия 335
  аннотированные на языке MGN 812
  из модуля django.views.generic.base 478
  из модуля django.views.generic.list 480
  наиболее важные типы коллекций 42
  MutableSequence и его суперклассов 427
  управляемых и дескрипторных 811
  упрощенная для collections.abc 51
  упрощенная для MutableMapping и его суперклассов 105
  упрощенная для MutableSetet4 и его суперклассов 127
  Sequence и связанных с ним абстрактных классов 416
  TomboList 438
Unicode
  база данных 165
  базовые кодировщики и декодировщики 140
  двухрежимный API 168
  диакритические знаки 159
  алгоритм
  упорядочивания (UCA) 164
  алгоритм упорядочивания Unicode 164
  канонические эквиваленты 155
  комбинирование символов 155
  обработка текстовых файлов 147
  проблемы кодирования и декодирования 141
  нормализация для надежного сравнения 155
  нотация литералов \\N{} 152
  основные сведения о байтах 137
  сворачивание регистра 158
  сравнение нормализованного текста 158
  символы и стандарт Unicode 136
  символы совместимости 156
  сортировка текста 162
  сэндвич Unicode 147
  SyntaxError, исключение 144
UnicodeDecodeError 143
UnicodeEncodeError 142
Union, тип 269
UserDict 118
uvloop 769

V

Vector2d
  пример класса 354
  полный код 365
  хешируемость 361
Vector, класс, многомерный
  доступ к динамическим атрибутам 390
  __format__ 399
  __hash__ и __eq__ 393
  применения 382
  протоколы и утиная типизация 385
  последовательности, допускающие срез 386
  совместимость с Vector2d 382

W
while/else, комбинация 638

Y
экземпляр дескриптора 812
yield, ключевое слово 240, 567, 568, 574, 598, 601, 622
yield from, выражение 590

A
абсолютное значение 38
агрегатные классы 484
Адаптер, паттерн 475
аргументы
  чисто именованные 242
  key 98
  фиксация с помощью functools.partial 248
  self 815, 832
арифметическая прогрессия, генератор 575
арифметические операторы 535
акцессоры 772, 807
анонимные функции 239, 252
аннотаций переменных
  синтаксис 186
аннотации переменных
  семантика 186
аннотации типов
  для асинхронных объектов 763
  аннотирование чисто позиционных и вариадических параметров 291
  достоинства и недостатки 254
  краткое введение 185
  и поддерживаемые операции 261
  обобщенных для классических сопрограмм 605
  несовершенная типизация и строгое тестирование 292
  постепенная типизация 255
  типы, пригодные для использования в 266
асинхронные генераторы 240, 721
асинхронное программирование
  аннотации типов для асинхронных объектов 763
  асинхронные контекстные менеджеры 729
  делегирование задач исполнителям 739
  итераторы и итерируемые объекты 751
  ловушка счетных функций 765
  миф о системах, ограниченных вводом-выводом 765
  написание асинхронных серверов 740
  объекты, допускающие ожидание 724
  пример использования asyncio 721
  проект Curio 760
  терминология 720
  улучшение асинхронного загрузчика 730
атрибуты
  виртуальные 772
  доступ к динамическим атрибутам 390
  закрытые и защищенные 368, 379
  защита и безопасность 379
  имена 778
  переопределение атрибутов класса 374
  проверка значений с помощью дескрипторов 811
  свойства и начальные затраты 378
  специальные 802
  удаление 800
  хранения 812, 816
  управляемые 812
  экземпляра 829

Б
байт-код, диассемблирование 306
блокировка
  определение термина 650
блоки with, match и else
  блоки else 638
  контекстные менеджеры и блоки with 613
блоки with, math и else
  сопоставление с образцом в lis.py 622
будущие объекты
  общие сведения 698
  практический пример 699
  определение термина 691

B
вариантность
  в типах Callable 289

инвариантные типы 514
ковариантные типы 516
контравариантные типы 516
кому интересна 514
нотация в других языках 526
эвристические правила 520
векторы
перегрузка оператора + 533
представление двумерных 37
ВерблюжьяНотация (CamelCase) 809, 878
виртуальные атрибуты 772
виртуальные подклассы 438, 440
встроенные функции 99
вызов по соиспользованию 219, 232
вызов по ссылке 231
вызываемые объекты
девять типов 240
в сочетании с iter() 560
пользовательские 241
выполнения этап 304, 417, 849
вычисляемые свойства 781
выборка связанных записей с помощью свойств 784
кеширование свойств 788
кеширование свойств с помощью functools 789
переопределение существующего атрибута свойством 787
создание управляемого данными атрибута 782

Г

генераторные функции
в стандартной библиотеке 578
реализация итерирования 567
определение термина 240
генераторы
делегирующие 590
генераторные сопрограммы 721
генераторные функции в стандартной библиотеке 578
арифметической прогрессии 575
асинхронные генераторные функции 752
реализация класса Sentence 567
ленивые 570
ключевое слово yield 568
когда использовать генераторные выражения 573
обобщенные итерируемые типы 596
примеры 568
сравнение с итераторами 574
субгенераторы 590
функции редуцирования итерируемого объекта 588
генераторные выражения 52, 55, 237, 573, 758
глобальная блокировка интерпретатора (GIL) 90, 650, 662, 673, 684
глубокая копия 218
гусиная типизация
ABC в стандартной библиотеке 427
виртуальные подклассы ABC 438
использование функции register 440
определение и использование ABC 430
определение термина 411
метод __subclasshook__ 441, 559
синтаксические детали ABC 435
создание подклассов ABC 426
структурная типизация 440

Д

двоичные последовательности 137
встроенные типы 137
разделение памяти 140
fromhex, метод класса 139
поддержка методов str 138
построение 139
способы отображения 138
декартово произведение, построение 54
декодирование
общие сведения 140
определение термина 137
Декоратор, паттерн 314, 331
декораторы и замыкания
динамическая область видимости 330
дополнение класса с помощью декоратора класса 847
в стандартной библиотеке Python 314
выполнение декораторов 302
замыкания в lis.py 636
реализация декоратора 312
регистрационные декораторы 304, 323

краткое введение в декораторы 301
композиция декораторов функций 435
назначение 300
параметризованные декораторы 322
объявление nonlocal 310
правила видимости переменных 304
пример замыкания 308
поведение декоратора 313
определение замыкания 310
основы замыканий 307
сравнение classmethod и staticmethod 357
декорирование имен 368
дерева обход 592
деструктуризация 66
дескрипторный класс 812
дескрипторы 188, 810. См. также дескрипторы атрибутов
проверяющий 828
дескрипторы атрибутов
дескрипторы данных 822
назначение 810
перезаписывание 825
переопределяющие и непереопределяющие 790, 820
методы как 826
проверка значений атрибутов 811
советы по использованию 828
строка документации и перехват удаления 829
терминология 811
диакритические знаки 159
динамические атрибуты и свойства
важные атрибуты и функции для работы с атрибутами 802
вычисляемые свойства 781
и виртуальные атрибуты 772
использование свойств для контроля значений 791
применение для обработки данных 773
программирование фабрики свойств 798
удаление 800
property, класс 794
динамическая область видимости 330
динамические протоколы 386, 415
динамический тип 266
дросселирование 733
дуализм функции и класса 809

Е

единица выполнения 648

З

загрузка файлов из веба
индикация хода выполнения 707, 713
обработка ошибок 711
параллельная и последовательная 692
запашки в коде 176, 199, 425
запоминание 315
замыкания. См. декораторы и замыкания;
защитное программирование 419

И

идентификаторы объектов 213
идиомы кодирования 409
изменяемые значения
вставка и обновление 108
именованные классы-образцы 202
импорта этап 302, 849
инвертированные индексы 741
индикаторы хода загрузки
реализация на основе процессов 655
реализация на основе потоков 652
реализация на основе сопрограмм 656
сравнение супервизоров 660
итераторы
асинхронные 751
ленивые последовательности 570
классы Sentence с методом __iter__ 564
и итерируемые объекты 561
обобщенные итерируемые типы 596
протокол последовательности 557
сравнение с генераторами 574
функция iter() 558
итерирование 556
неявная природа 35
с помощью генераторных функций 567
итерируемые объекты 559
асинхронные 751

распаковка 61
и итераторы 561
функции редуцирования 588
интернирование 227
интерфейсы. См. также гусиная типизация; протоколы
ABC (абстрактный базовый класс) 426
Container 42
роль в объектно-ориентированном программировании 411
карта типизации 412
протоколы как неформальные интерфейсы 407
Iterable 42
Sequence 415
Sized 42
явные 483
инфиксные операторы 528, 540
обработка исключений 538
операнды 534
особый механизм диспетчеризации 534

К

карта типизации 412, 459
канонические эквиваленты 155
кеширование
и дескрипторы атрибутов 828
и слабые ссылки 226
крокозябры, определение термина 143
кладистика 423
классы
агрегатные 484
дескрипторные 810
реализация обобщенного класса 511
как вызываемые объекты 240
как объекты 835, 877
недокументированные 526
примеси 477, 484
нотация MGN 813
классы протоколов 386
ключевые слова
as 67
await 724
зарезервированные 631
lambda 239
nonlocal 311, 634
yield 240, 567, 574, 598, 601, 622
ключи
автоматическая обработка отсутствующих 111
обработка отсутствия 108
практические последствия внутреннего устройства класса dict 122
преобразование нестроковых ключей в тип str 118
постоянное хранилище для отображения 117
сортировка по нескольким 246
сохранение порядка вставки 43, 868
хешируемость 105
ковариантность. См. вариантность
кодек 140
кодировка
проблемы кодирования и декодирования 141
определение 136
по умолчанию 150
основные сведения 141
UCS-2 141
UTF-8 145
UTF-8-SIG 147
кодовая позиция 136
кортежи
аннотации типа 273
распаковка 58
как неизменяемые списки 58
классические именованные 181
о природе 96
относительная неизменяемость 214, 226
сравнение со списками 60
упрощенная диаграмма размещения в памяти 50
typing.NamedTuple 184
коллекции
как итерируемые объекты 556
колода карт, пример 33
Команда, паттерн проектирования 345
композиция 483
композиция декораторов 316
композиция объектов 483
контейнерные последовательности 97
контекстные менеджеры
демонстрации 613
асинхронные 729

асинхронные генераторы
в качестве 756
@contextmanager, декоратор 617
интерфейс 613
назначение 613
необычные применения 613
скобочные в Python 3.10 616
утилиты contextlib 617
контравариантность. См. вариантность
конкатенация 76
конкурентность
загрузка с применением concurrent.futures 696
запуск задач с помощью concurrent.futures 701
индикация хода выполнения 707
использование futures.as_completed 713
обработка ошибок 707, 711
примеры 692
сравнение параллельного и последовательного скриптов 693
тестирование параллельных клиентов 707
конкурентные исполнители
загрузка с индикацией хода выполнения и обработкой ошибок 707
конкурентная загрузка из веба 692
Executor.map 704
кооперативное множественное наследование 471
копирование
глубокое 218
поверхностное 215
коэффициент Отиаи 382

Л

лексическая область видимости 331
ленивое вычисление 552, 570
локаль, установка 163
Лундха рецепт рефакторинга лямбда-выражений 239

М

магические методы 33
массивы 50, 83
метаклассы
демонстрация работы 860
реализация Checked с помощью 864
предостережения 868
пример 857
определение термина 837
полезные применения 872
метаобъекты 47
метапрограммирование 302
метапрограммирование классов
дополнение класса с помощью декоратора 847
достоинства и недостатки 834
встроенная фабрика классов 836
реализация Checked с помощью метакласса 864
классы как объекты 835
метод __prepare__ 870
проблемы метаклассов 868
полезные применения метаклассов 872
__init_subclass__ 840
фабрика классов 837
этап импорта и этап выполнения 849
методы
встроенные 240
как дескрипторы 826
как вызываемые объекты 240
определение термина 772
мини-язык спецификации формата 359, 399
множества. См. также словари и множества
литеральные 125
практические последствия внутреннего устройства класса set 126
операции над 127
множественное включение 126
теория множеств 123
множественное наследование. См. также наследование
на практике 475
многоточие (...) 35, 74
многоядерная обработка
веб-разработка на стороне сервера и на мобильных устройствах 676
распределенные очереди задач 680
наука о данных 675
системное администрирование 674
WSGI-серверы приложений 678

модели конкурентности
влияние GIL 662
конкурентная программа Hello World 652
и многоядерные процессоры 673
основные сведения 646
пулы процессов 665
структурная конкурентность 761
терминология 648
модель данных
использование специальных методов 36
методы __getitem__ и __len__ 33
общие сведения 32
поведение __len__ 45
сводка специальных методов 43

Н

накопительное среднее, вычисление 308, 599
наследование
в разных языках 490
виртуальные подклассы ABC 438
встроенным типам 465
рекомендации 482
классы-примеси 473
и композиция 382
обобщенным базовым классам 426, 435
практические примеры 475
множественное наследование и порядок разрешения методов 468
функция super() 463
начальное значение 106
несовершенная типизация 292
неявное преобразование 135
номинальная типизация 262

О

обобщенные коллекции
и аннотации типов 270
параметризованные обобщенные типы и TypeVar 280
обобщенные отображения 275
обобщенные статические протоколы 520
обобщенные функции 318
область видимости
глобальная область видимости модуля 306
динамическая и лексическая 330
внутри включений и генераторных выражений 53
локальная область видимости функции 306
правила видимости переменных 304
обработка данных
гибкое создание объектов 779
исследование JSON-подобных данных 775
применение динамических атрибутов 773
проблема недопустимого имени атрибута 778
объектная модель 46
объекты
в духе Python 352
гибкое создание 779
вызываемые 240, 560
изменяемые 220, 230
классы как 835
итерируемые 559
обращение с функцией как с объектом 235
одиночки 214
полноправные 234, 876
пользовательские вызываемые 241
хешируемые 105, 361
уничтожение 231
объекты в духе Python. См. также объекты
альтернативный конструктор 356
закрытые и защищенные атрибуты 368
представления объекта 353
пример класса Vector2d 354
поддержка позиционного сопоставления с образцом 363
полный код класса Vector2d 365
сравнение classmethod и staticmethod 357
создание собственных классов 352
хешируемый класс Vector2d 361
форматирование при выводе 358
экономия памяти с помощью __slots__ 370
опережающей ссылки проблема 508
операторы составного присваивания 78, 545

оптимизация хвостовой рекурсии (TCO) 643
отображения
  автоматическая обработка отсутствующих ключей 111
  распаковка 101
  не зависящие от регистра 473
  обзор методов 106
  неизменяемые 120
  объединение 102
  сопоставление с отображением-образцом 102
  стандартный API 105
очереди
  реализация 92
  распределенные очереди задач 680
  deque (двусторонняя очередь) 83
  определение термина 649

П

параллелизм 646, 648
параллельное присваивание 62
параметризованные типы 514
параметры
  чисто именованные 243
  чисто позиционные 244
  аннотирование чисто позиционных и вариадических параметров 291
  изменяемые 220, 222
  интроспекция параметров функций 235
  передача 231
параметры функций 219
параметры функций, интроспекция 235
партизанское латание 417, 460, 825
паттерны проектирования 333
перегруженные сигнатуры 492
перегрузка операторов 528
  достоинства 529, 552
  имена методов инфиксных операторов 541
  инфиксные операторы 528, 541
  использование @ как инфиксного оператора 540
  недостатки 552
  операторы сравнения 542
  операторы составного присваивания 545
  сложение векторов 533
  унарные операторы 530
  умножение на скаляр 538
перегрузка функций и методов 319
переработка паттерна Стратегия
  дополнение декоратором 343
  выбор наилучшей стратегии 341
  классический паттерн 334
  паттерн Команда 345
  поик стратегий в модуле 342
  функционально-ориентированная стратегия 338
переменные
  логика поиска 311
  метафора этикеток и ящиков 210
  свободные 309
  только для инициализации 196
платформенные сопрограммы
  определение термина 720
  сравнение с асинхронными генераторами 757
  функции, определенные с помощью async def 240
подмешанные методы 475
подсчет ссылок 225
подтипизация по поведению 268
позиционное сопоставление с образцом 363
позиционные классы-образцы 204
позиционные параметры 242
полноправные объекты 234, 876
поразрядные операторы 44, 530
порядок разрешения методов 440, 473
последовательности
  единообразная обработка 48
  альтернативы списку 83
  допускающие срез 386
  доступ к динамическим атрибутам 390
  встроенные 49
  распаковка 61
  изменяемые 50, 418
  кортежи 57
  контейнерные 49, 97
  использование + и * 76
  __format__ 399
  __hash__ и __eq__ 393
  неизменяемые 50
  протоколы и утиная типизация 385
  плоские 49, 97
  получение среза 72

построение 51
совместимость с Vector2d 382
сортировка 81
списковое включение и генераторные выражения 51
сопоставление с образцом 64
стратегия реализации вектора 382
последовательности байтов 145
построители классов данных
аннотации типов 185
@dataclass 190
класс данных как признак кода с душком 199
классические именованные кортежи 181
основные возможности 179
сопоставление с экземплярами классов – образцами 201
типизированные именованные кортежи 184
поток
влияние GIL 663
реализация индикатора хода загрузки 652
нежелательности 717
определение термина 649
улучшение асинхронного загрузчика 730
представления областей памяти 86
представления с ограниченной длиной 383
приведение типов 505
принцип единообразного доступа 772, 806, 807
принцип быстрого отказа 108, 398, 419
программ проверки типов для Python 256
процесс
запуск с помощью concurrent.futures 701
определение термина 648
протоколы. См. также интерфейсы
числовые 453
динамическая природа 419
динамические 386
достоинства 418
защитное программирование 419
значения слова 413
реализация во время выполнения 417
реализация обобщенного статического протокола 520
как неформальные интерфейсы 407
и утиная типизация 385
последовательности 386, 414, 415, 557
статические 386, 443
псевдонимия 212
пулы процессов
код проверки на простоту для многоядерной машины 668
интерпретация времени работы 667
пример задачи 665

P

распаковка
вложенных объектов 63
использование * в вызовах функций и литеральных последовательностях 63
использование * для выборки лишних элементов 62
последовательностей и итерируемых объектов 61
отображений 101
распределенные очереди задач 680
регулярные выражения, str и bytes 168
ромбовидного наследования проблема 468

C

сборка мусора 224, 231
серверы
класс HTTPServer 475
класс ThreadingHTTPServer 476
написание асинхронных 740
тестовые 710
шлюзовой интерфейс веб-серверов (WSGI) 678
шлюзовой интерфейс асинхронных серверов (ASGI) 680
TCP-сервер 746
свободные переменные 309, 331, 636
свойства
достоинства 876
выборка связанных записей 784
и атрибуты экземпляров 795
общие сведения 794
строки документации 797
фабрики свойств 798

сворачивание регистра 158
семафоры 733
сетевой ввод-вывод
  загрузка с индикацией хода выполнения и обработкой ошибок 707
  загрузка с применением asyncio 725
  загрузка с применением concurrent.futures 696
  миф о системах, ограниченных вводом-выводом 765
  скрипт последовательной загрузки 694
  улучшение асинхронного загрузчика 730
сравнения операторы 44, 213, 530, 542
срезка
  демонстрация 387
  исключение последнего элемента среза 73
  объекты среза 73
  присваивание срезу 75
  многомерные срезы и многоточие 74
  последовательности, допускающие срез 386
синглтон 214
символы
  числовое значение 167
  кодирование и декодирование 137
  определение 136
  поиск по имени 165
  совместимости 156
синтаксический сахар 133
система постепенной типизации
  чтение аннотаций типов во время выполнения 508
  абстрактные базовые классы 276
  реализация обобщенного класса 511
  реализация обобщенного статического протокола 520
  и вариантность 514
  перегруженные сигнатуры 492
  параметризованные обобщенные типы и TypeVar 280
  обобщенные коллекции 270
  обобщенные отображения 275
  на практике 256
  приведение типов 505
  простые типы и классы 269
  поддержка унаследованных типов и нерекомендуемые типы коллекций 271
  отношения является подтипом и совместим с 267
  тип Any 266
  тип Callable 288
  тип Iterable 278
  тип NoReturn 291
  типы кортежей 273
  типы Optional и Union 269
  TypedDict 498
системное администрирование 674
системы, ограниченные вводом-выводом 765
слабые ссылки 226
словари и множества
  вариации на тему dict 115
  автоматическая обработка отсутствующих ключей 111
  внутреннее устройство словарей 100
  неизменяемые отображения 120
  представления словаря 121
  практические последствия внутреннего устройства класса dict 122
  практические последствия внутреннего устройства класса set 126
  операции над множествами 127
  современный синтаксис словарей 100
  сопоставление с отображением-образцом 102
  стандартный API типов отображений 105
  теоретико-множественные операции над представлениями словарей 129
  теория множеств 123
словарное включение (dictcomp) 100
сложение на месте 78, 548
совместимости символы 156
специальные атрибуты 802
специальные методы
  для работы с атрибутами 804
  булево значение пользовательского типа 41
  API коллекций 42
  вызов 36

__getitem__ и __len__ 33
назначение 32
неявная природа 37
преимущества 35
пример реализации 39
сводка 43
соглашение об именовании 32
строковое представление 40
эмуляция числовых типов 37
сортировка 81, 98
списки
альтернативы 83
кортежи как неизменяемые списки 60
поверхностное копирование 215
многострочные 52
построение списка списков 76
сравнение list.sort и встроенной функции sorted 81
смешанные типы 97
списковое включение (listcomp)
генерация декартова произведения 54
вложенные списки 76
асинхронное 758
замечание о синтаксисе 52
и генераторные выражения 55
как альтернатива функциям map и filter 237
локальная область видимости 53
и удобочитаемость 52
построение последовательностей 51
сопрограммы
генераторные 721
влияние GIL 663
возврат значения 601
аннотации обобщенных типов 605
вычисление накопительного среднего 599
классические 598
индикатор хода выполнения 656
определение термина 649
типы 720
сопоставление с образцом
вариадические параметры 291
деструктуризация 66
в интерпретаторе lis.py 622
предложение match/case 64
*_, символ 68
символ _ 66
с кортежами и списками 66
с отображением-образцом 102
с экземплярами классов-образцами 201
учет информации о типе 67
составные объекты 96
состязание 650
ссылки на объекты
глубокое копирование 218
del и сборка мусора 224
и неизменяемость 226
параметры функций как ссылки 219
метафора ящиков и этикеток 210
поверхностное копирование 215
псевдонимия 212
статическая типизация 411
статическая утиная типизация 288, 411, 487
статические протоколы
ABC из пакета numbers и числовые протоколы 453
допускающие проверку во время выполнения 444
реализация обобщенного статического протокола 520
рекомендации по проектированию 451
расширение 452
ограничения протоколов, допускающих проверку во время выполнения 447
проектирование 450
поддержка 448
определение 415
сравнение с динамическими 386
типизированная функция double 443
Стратегия, паттерн 334
строгое тестирование 292
строки
двухрежимный API 168
представление с помощью специальных методов 40
сортировка по умолчанию 82
структурная конкурентность 761
структурная типизация 440
структуры данных
последовательности 48
текст и байты 135
субгенераторы 590

схема дублинского ядра 197
счетчик ссылок 231

Т

текстовые файлы
  кодирование и декодирование 147
теория множеств
  литеральные множества 125
  математические операции 128
  операторы сравнения множеств 129
  множественное включение 126
  хешируемость элементов 124
типизированная функция double 443

У

умножение
  вектора на скаляр 38, 538
унарные операторы 530
управляемые атрибуты 812
управляемые классы 812
управляемые экземпляры 812
утиная типизация 108, 262, 298, 385, 408, 411, 446

Ф

фактические параметры-типы 514
фильтрующие генераторные функции 578
формальные параметры-типы 514
функции
  декораторы и замыкания 435
  генераторные 240
  abs, встроеннная функция 38
  диассемблирование байт-кода 306
  анонимные 252, 307
  встроенные 99, 240, 803
  apply() 237
  dir([object]) 803
  double() 443
  как полноправные объекты 876
  filter() 53
  format() 358
  getattr(object, name[, default]) 803
  globals() 342
  hasattr() 803
  обобщенные с одиночной диспетчеризацией 318
  id() 213
  пользовательские 240
  iter() 559
  len() 34
  map() 53
  map, filter и reduce 237
  max() 494
  random.choice() 35
  repr() 353
  setattr(object, name, value) 803
  sorted() 81
  str() 353
  super() 392, 463
  vars([object]) 803
  zip() 397
функции, аннотации типов
  аннотирование чисто позиционных и вариадических параметров 291
  достоинства и недостатки 254
  несовершенная типизация и строгое тестирование 292
  постепенная типизация 255
  типы, пригодные для использования в аннотациях 266
функции как полноправные объекты. См. также декораторы и замыкания
  гибкая обработка параметров 242
  анонимные функции 239
  вызываемые объекты 240
  обращение с функцией как с объектом 235
  определение термина 234
  пользовательские вызываемые типы 241
  функции высшего порядка 236
функциональное программирование
  пакеты для 245
  на Python 251
функции редуцирования 588

Х

хешируемость
  определение 105
хешируемые объекты 361
хеш-таблицы 99

Ч

числовая башня 278
числовые протоколы 453
числовые типы
  проверка на конвертируемость 445, 455

поддержка 454
хешируемость 106
эмуляция 37
эмуляция с помощью специальных методов 37
чисто именованные аргументы 242
чисто хвостовая рекурсия (PTC) 643

Ш

шлюзовой интерфейс веб-серверов (WSGI) 678
шлюзовой интерфейс асинхронных серверов (ASGI) 680

Э

эмодзи
в Музее современного искусства 173
и консольный шрифт 152
нарастание проблем 175
полнота поддержки 166
поиск символов по имени 165
создание 136

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «КТК Галактика» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, пр. Андропова д. 38 оф. 10.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.galaktika-dmk.com**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Лусиану Рамальо

Python – к вершинам мастерства

Лаконичное и эффективное программирование

Второе издание

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 72,96. Тираж 200 экз.

Веб-сайт издательства: *www.dmkpress.com*

O'REILLY®

Не тратьте зря времени, пытаясь подогнать Python под способы программирования, знакомые вам по другим языкам. Python настолько прост, что вы очень быстро освоите его в общих чертах, но для создания эффективных современных программ требуются более глубокие знания.

Второе издание книги позволит вам использовать возможности Python 3 в полной мере, обратив себе на пользу лучшие идеи. Автор рассказывает о базовых средствах языка, о его библиотеках и учит писать более краткий, быстрый и удобочитаемый код. Вы узнаете о том, как применять идиоматические средства Python 3, выходящие за рамки вашего предыдущего опыта.

Новое издание состоит из пяти частей, которые можно рассматривать как пять отдельных книг.

- **Структуры данных:** последовательности, словари, множества, Unicode и классы данных.
- **Функции как объекты:** полноправные функции, относящиеся к ним паттерны проектирования, а также аннотации типов в объявлениях функций.
- **Объектно-ориентированные идиомы:** композиция, наследование, классы-примеси, интерфейсы, перегрузка операторов, протоколы и дополнительные статические типы.
- **Поток управления:** контекстные менеджеры, генераторы, сопрограммы, async/await, пулы процессов и потоков.
- **Метапрограммирование:** свойства, дескрипторы атрибутов, декораторы классов и новые средства метапрограммирования классов, позволяющие заменить или упростить метаклассы.

«Моя палочка-выручалочка, к которой я обращаюсь, когда нужно найти подробное объяснение и примеры использования какого-нибудь средства Python. Методика преподавания и изложения материала, свойственная Лусиану, вызывает восхищение. Прекрасная книга для программистов, уже имеющих какой-то опыт и желающих пополнить свой багаж знаний».

Кэрол Уиллинг,
член Руководящего совета Python (2020–2021)

«Это не обычная сухая книга по кодированию, а кладезь полезных протестированных примеров, сдобренных изрядной толикой юмора. Нам с коллегами эта отлично написанная и увлекательная книга позволила поднять свои навыки программирования на Python на следующий уровень».

Мария Маккинли,
старший программист

Лусиану Рамальо – главный консультант в компании Thoughtworks и член фонда Python Software Foundation.

ISBN 978-5-97060-885-2

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф