Авторитетный курс объектно-ориентированного программирования

5-е издание

Изучаем Python

Том 2

ДИАЛЕКТИКА

O'REILLY

Марк Лутц

5-е издание

Изучаем Python

Том 2

FIFTH EDITION

Learning Python

Mark Lutz

Beijing • Boston • Farnham • Sebastopol • Tokyo

5-е издание

Изучаем Python

Том 2

Марк Лутц

Москва · Санкт-Петербург
2020

ББК 32.973.26-018.2.75
Л86
УДК 681.3.07

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *Ю.Н. Артеменко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Лутц, Марк.

Л86 Изучаем Python, том 2, 5-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2020. — 720 с. : ил. — Парал. тит. англ.

ISBN 978-5-907144-53-8 (рус., том 2)
ISBN 978-5-907144-51-4 (рус., многотом.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly&Associates.

Authorized Russian translation of the English edition of *Learning Python, 5th Edition* (ISBN 978-1-449-35573-9) © 2013 by Mark Lutz.

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

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Марк Лутц

Изучаем Python, том 2
5-е издание

Подписано в печать 25.11.2019. Формат 70×100/16.
Гарнитура Times.
Усл. печ. л. 58,05. Уч.-изд. л. 47,3.
Тираж 1000 экз. Заказ № 10632.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-53-8 (рус., том 2)
ISBN 978-5-907144-51-4 (рус., многотом.)

ISBN 978-1-449-35573-9 (англ.)

© 2020, ООО "Диалектика"

© 2013 by Mark Lutz

Оглавление

Предисловие 17

Часть VI. Классы и объектно-ориентированное программирование 19

ГЛАВА 26. Объектно-ориентированное программирование: общая картина 20

ГЛАВА 27. Основы написания классов 34

ГЛАВА 28. Более реалистичный пример 54

ГЛАВА 29. Детали реализации классов 96

ГЛАВА 30. Перегрузка операций 123

ГЛАВА 31. Проектирование с использованием классов 169

ГЛАВА 32. Расширенные возможности классов 216

Часть VII. Исключения и инструменты 315

ГЛАВА 33. Основы исключений 316

ГЛАВА 34. Детали обработки исключений 327

ГЛАВА 35. Объекты исключений 357

ГЛАВА 36. Проектирование с использованием исключений 375

Часть VIII. Более сложные темы 399

ГЛАВА 37. Unicode и байтовые строки 400

ГЛАВА 38. Управляемые атрибуты 455

ГЛАВА 39. Декораторы 504

ГЛАВА 40. Метаклассы 590

ГЛАВА 41. Все хорошее когда-нибудь заканчивается 644

Часть IX. Приложения 653

Приложение А. Установка и конфигурирование 654

Приложение Б. Запускающий модуль Windows для Python 668

Приложение В. Изменения в Python и настоящая книга 677

Приложение Г. Решения упражнений, приводимых в конце частей 692

Предметный указатель 709

Содержание

Предисловие 17

Часть VI. Классы и объектно-ориентированное программирование 19

ГЛАВА 26. Объектно-ориентированное программирование: общая картина 20

Для чего используются классы? 21
Объектно-ориентированное программирование с высоты птичьего полета 22
    Поиск в иерархии наследования 23
    Классы и экземпляры 25
    Вызовы методов 25
    Создание деревьев классов 26
    Перегрузка операций 28
    Объектно-ориентированное программирование — это многократное использование кода 29
Резюме 32
Проверьте свои знания: контрольные вопросы 32
Проверьте свои знания: ответы 33

ГЛАВА 27. Основы написания классов 34

Классы генерируют множество объектов экземпляров 34
    Объекты классов обеспечивают стандартное поведение 35
    Объекты экземпляров являются конкретными элементами 35
    Первый пример 36
Классы настраиваются через наследование 38
    Второй пример 39
    Классы являются атрибутами в модулях 41
Классы могут перехватывать операции Python 42
    Третий пример 43
    Для чего используется перегрузка операций? 45
Простейший в мире класс Python 46
    Снова о записях: классы или словари 49
Резюме 51
Проверьте свои знания: контрольные вопросы 51
Проверьте свои знания: ответы 52

ГЛАВА 28. Более реалистичный пример 54

Шаг 1: создание экземпляров 55
    Написание кода конструкторов 55
    Тестирование в ходе дела 56
    Использование кода двумя способами 58
Шаг 2: добавление методов, реализующих поведение 59
    Написание кода методов 61
Шаг 3: перегрузка операций 63
    Реализация отображения 63

Шаг 4: настройка поведения за счет создания подклассов 65
Написание кода подклассов 66
Расширение методов: плохой способ 66
Расширение методов: хороший способ 67
Полиморфизм в действии 69
Наследование, настройка и расширение 70
Объектно-ориентированное программирование: основная идея 71
Шаг 5: настройка конструкторов 72
Объектно-ориентированное программирование проще, чем может казаться 73
Другие способы комбинирования классов 74
Шаг 6: использование инструментов интроспекции 77
Специальные атрибуты класса 78
Обобщенный инструмент отображения 79
Атрибуты экземпляра или атрибуты класса 81
Размышления относительно имен в классах инструментов 82
Финальная форма классов 83
Шаг 7 (последний): сохранение объектов в базе данных 84
Модули `pickle`, `dbm` и `shelve` 85
Сохранение объектов в базе данных `shelve` 86
Исследование хранилища `shelve` в интерактивной подсказке 87
Обновление объектов в хранилище `shelve` 89
Указания на будущее 91
Резюме 93
Проверьте свои знания: контрольные вопросы 93
Проверьте свои знания: ответы 94

ГЛАВА 29. Детали реализации классов 96

Оператор `class` 96
Общая форма 97
Пример 97
Методы 99
Пример метода 100
Вызов конструкторов суперклассов 101
Другие возможности вызова методов 101
Наследование 102
Построение дерева атрибутов 102
Специализации унаследованных методов 104
Методики связывания классов 104
Абстрактные суперклассы 106
Пространства имен: заключение 108
Простые имена: глобальные, если не выполнено их присваивание 109
Имена атрибутов: пространства имен объектов 109
“Дзен” пространств имен: присваивания классифицируют имена 110
Вложенные классы: снова о правиле областей видимости LEGB 112
Словари пространств имен: обзор 114
Связи между пространствами имен: инструмент подъема по дереву 117
Снова о строках документации 119
Классы или модули 120

Резюме 121
Проверьте свои знания: контрольные вопросы 121
Проверьте свои знания: ответы 122

ГЛАВА 30. Перегрузка операций 123

Основы 123
Конструкторы и выражения: `__init__` и `__sub__` 124
Распространенные методы перегрузки операций 124
Индексирование и нарезание: `__getitem__` и `__setitem__` 127
Перехват срезов 127
Нарезание и индексирование в Python 2.X 129
Но метод `__index__` в Python 3.X не имеет отношения к индексированию! 130
Итерация по индексам: `__getitem__` 130
Итерируемые объекты: `__iter__` и `__next__` 131
Итерируемые объекты, определяемые пользователем 132
Множество итераторов в одном объекте 135
Альтернативная реализация: `__iter__` плюс `yield` 138
Членство: `__contains__`, `__iter__` и `__getitem__` 142
Доступ к атрибутам: `__getattr__` и `__setattr__` 145
Ссылка на атрибуты 146
Присваивание и удаление атрибутов 147
Другие инструменты управления атрибутами 148
Эмуляция защиты атрибутов экземпляра: часть 1 149
Строковое представление: `__repr__` и `__str__` 150
Для чего используются два метода отображения? 151
Замечания по использованию отображения 152
Использование с правой стороны и на месте: `__radd__` и `__iadd__` 153
Правостороннее сложение 154
Сложение на месте 157
Выражения вызовов: `__call__` 158
Функциональные интерфейсы и код, основанный на обратных вызовах 160
Сравнения: `__lt__`, `__gt__` и другие 162
Метод `__cmp__` в Python 2.X 163
Булевские проверки: `__bool__` и `__len__` 163
Булевские методы в Python 2.X 164
Уничтожение объектов: `__del__` 166
Замечания относительно использования деструкторов 166
Резюме 167
Проверьте свои знания: контрольные вопросы 168
Проверьте свои знания: ответы 168

ГЛАВА 31. Проектирование с использованием классов 169

Python и объектно-ориентированное программирование 169
Полиморфизм означает интерфейсы, а не сигнатуры вызовов 170
Объектно-ориентированное программирование и наследование: отношения “является” 171
Объектно-ориентированное программирование и композиция: отношения “имеет” 173

Снова об обработчиках потоков данных 174
Объектно-ориентированное программирование и делегирование: промежуточные объекты-оболочки 178
Псевдозакрытые атрибуты классов 180
Обзор корректировки имен 180
Для чего используются псевдозакрытые атрибуты? 181
Методы являются объектами: связанные или несвязанные методы 183
Несвязанные методы являются функциями в Python 3.X 185
Связанные методы и другие вызываемые объекты 187
Классы являются объектами: обобщенные фабрики объектов 190
Для чего используются фабрики? 191
Множественное наследование: “подмешиваемые” классы 192
Реализация подмешиваемых классов отображения 193
Другие темы, связанные с проектированием 214
Резюме 214
Проверьте свои знания: контрольные вопросы 215
Проверьте свои знания: ответы 215

ГЛАВА 32. Расширенные возможности классов 216

Расширение встроенных типов 217
Расширение типов путем внедрения 217
Расширение типов путем создания подклассов 218
Модель классов “нового стиля” 220
Что нового в новом стиле? 221
Изменения в классах нового стиля 222
Процедура извлечения атрибутов для встроенных операций пропускает экземпляры 224
Изменения модели типов 229
Все классы являются производными от `object` 232
Изменение ромбовидного наследования 234
Дополнительные сведения о MRO: порядок распознавания методов 238
Пример: отображение атрибутов на источники наследования 241
Расширения в классах нового стиля 246
Слоты: объявления атрибутов 247
Свойства: средства доступа к атрибутам 256
Метод `__getattribute__` и дескрипторы: инструменты для работы с атрибутами 259
Другие изменения и расширения классов 260
Статические методы и методы классов 261
Для чего используются специальные методы? 261
Статические методы в Python 2.X и 3.X 262
Альтернативы для статических методов 264
Использование статических методов и методов класса 265
Подсчет экземпляров с помощью статических методов 267
Подсчет экземпляров с помощью методов классов 268
Декораторы и метаклассы: часть 1 271
Основы декораторов функций 272
Первый взгляд на декораторы функций, определяемые пользователем 273
Первый взгляд на декораторы классов и метаклассы 275

Дополнительные сведения 277
Встроенная функция super: для лучшего или для худшего? 277
Продолжительные дебаты относительно super 277
Традиционная форма вызова методов суперкласса: переносимая, универсальная 279
Базовое использование встроенной функции super и связанные с ней компромиссы 279
Положительные стороны super: изменения деревьев и координирование 285
Изменения классов во время выполнения и super 286
Кооперативная координация вызовов методов при множественном наследовании 287
Сводка по super 299
Затруднения, связанные с классами 300
Изменение атрибутов классов может иметь побочные эффекты 301
Модификация изменяемых атрибутов классов тоже может иметь побочные эффекты 302
Множественное наследование: порядок имеет значение 303
Области видимости в методах и классах 304
Другие затруднения, связанные с классами 305
Еще раз о KISS: чрезмерно большое количество уровней 306
Резюме 307
Проверьте свои знания: контрольные вопросы 307
Проверьте свои знания: ответы 307
Проверьте свои знания: упражнения для части VI 309

Часть VII. Исключения и инструменты 315

ГЛАВА 33. Основы исключений 316

Для чего используются исключения? 316
Роли, исполняемые исключениями 317
Исключения: краткая история 318
Стандартный обработчик исключений 318
Перехват исключений 320
Генерация исключений 321
Исключения, определяемые пользователем 321
Действия при завершении 322
Резюме 325
Проверьте свои знания: контрольные вопросы 326
Проверьте свои знания: ответы 326

ГЛАВА 34. Детали обработки исключений 327

Оператор try/except/else 327
Как работают операторы try 328
Конструкции оператора try 329
Конструкция else оператора try 332
Пример: стандартное поведение 333
Пример: перехват встроенных исключений 334

Оператор try/finally 335
Пример: написание кода действий при завершении с помощью try/finally 336
Унифицированный оператор try/except/finally 337
Унифицированный синтаксис оператора try 338
Комбинирование finally и except за счет вложения 339
Пример унифицированного оператора try 339
Оператор raise 341
Генерация исключений 342
Области видимости и переменные except в try 343
Распространение исключений с помощью raise 344
Сцепление исключений в Python 3.X: raise from 345
Оператор assert 347
Пример: улавливание нарушений ограничений (но не ошибок!) 348
Диспетчеры контекстов with/as 349
Базовое использование 350
Протокол управления контекстами 351
Множество диспетчеров контекстов в Python 3.1, 2.7 и последующих версиях 353
Резюме 355
Проверьте свои знания: контрольные вопросы 355
Проверьте свои знания: ответы 355

ГЛАВА 35. Объекты исключений 357

Исключения: назад в будущее 358
Строковые исключения канули в лету! 358
Исключения на основе классов 359
Реализация классов исключений 360
Для чего используются иерархии исключений? 362
Встроенные классы исключений 365
Категории встроенных исключений 366
Стандартный вывод и состояние 367
Специальное отображение при выводе 369
Специальные данные и поведение 371
Предоставление деталей исключения 371
Предоставление методов исключений 372
Резюме 373
Проверьте свои знания: контрольные вопросы 373
Проверьте свои знания: ответы 374

ГЛАВА 36. Проектирование с использованием исключений 375

Вложение обработчиков исключений 375
Пример: вложение в потоке управления 377
Пример: синтаксическое вложение 377
Идиомы исключений 379
Прерывание множества вложенных циклов: “безусловный переход” 379
Исключения не всегда являются ошибками 380
Функции могут сигнализировать об условиях с помощью raise 381
Закрытие файлов и серверных подключений 382
Отладка с помощью внешних операторов try 383
Выполнение внутрипроцессных тестов 383

Дополнительные сведения о функции sys.exc_info 384
Отображение сообщений об ошибках и трассировок 385
Советы по проектированию с использованием исключений и связанные с ними затруднения 386
Что должно быть помещено внутрь операторов try? 386
Перехват слишком многого: избегайте использования пустой конструкции except и конструкции except Exception 387
Перехват чересчур малого: используйте категории на основе классов 389
Сводка по базовому языку 390
Комплект инструментов Python 390
Инструменты для разработки, ориентированные на более крупные проекты 391
Резюме 395
Проверьте свои знания: контрольные вопросы 396
Проверьте свои знания: ответы 396
Проверьте свои знания: упражнения для части VII 396

Часть VIII. Более сложные темы 399

ГЛАВА 37. Unicode и байтовые строки 400

Изменения строк в Python 3.X 401
Основы строк 402
Схемы кодирования символов 402
Хранение строк Python в памяти 405
Типы строк Python 406
Текстовые и двоичные файлы 408
Написание базовых строк 409
Строковые литералы Python 3.X 410
Строковые литералы Python 2.X 412
Преобразования строковых типов 412
Написание строк Unicode 414
Написание текста ASCII 414
Написание текста, отличающегося от ASCII 415
Кодирование и декодирование текста, отличающегося от ASCII 416
Другие схемы кодирования 417
Байтовые строковые литералы: закодированный текст 418
Преобразования между кодировками 420
Кодирование строк Unicode в Python 2.X 420
Объявления кодировок в файлах исходного кода 424
Использование объектов bytes в Python 3.X 425
Вызовы методов 425
Операции над последовательностями 426
Другие способы создания объектов bytes 427
Смешивание строковых типов 428
Использование объектов bytearray в Python 3.X/2.6+ 428
Объекты bytearray в действии 429
Сводка по строковым типам Python 3.X 431
Использование текстовых и двоичных файлов 431
Основы текстовых файлов 432
Текстовый и двоичный режимы в Python 2.X и 3.X 433

Несоответствия типов и содержимого в Python 3.X 434
Использование файлов Unicode 435
Чтение и запись данных Unicode в Python 3.X 436
Обработка маркера BOM в Python 3.X 437
Файлы Unicode в Python 2.X 440
Имена файлов и потоки данных Unicode 441
Другие изменения инструментов для обработки строк в Python 3.X 442
Модуль re для сопоставления с образцом 442
Модуль struct для работы с двоичными данными 444
Модуль pickle для сериализации объектов 446
Инструменты для разбора XML 447
Резюме 451
Проверьте свои знания: контрольные вопросы 452
Проверьте свои знания: ответы 452

ГЛАВА 38. Управляемые атрибуты 455

Для чего используются управляемые атрибуты? 455
Вставка кода для запуска при доступе к атрибутам 456
Свойства 457
Основы 458
Первый пример 458
Вычисляемые атрибуты 459
Реализация свойств с помощью декораторов 460
Дескрипторы 462
Основы 462
Первый пример 465
Вычисляемые атрибуты 467
Использование информации состояния в дескрипторах 468
Связь между свойствами и дескрипторами 471
__getattr__ и __getattribute__ 473
Основы 474
Первый пример 477
Вычисляемые атрибуты 478
Сравнение __getattr__ и __getattribute__ 480
Сравнение методик управления 481
Перехват атрибутов для встроенных операций 484
Пример: проверка достоверности атрибутов 491
Использование свойств для проверки достоверности 492
Использование дескрипторов для проверки достоверности 494
Использование __getattr__ для проверки достоверности 498
Использование __getattribute__ для проверки достоверности 500
Резюме 501
Проверьте свои знания: контрольные вопросы 502
Проверьте свои знания: ответы 502

ГЛАВА 39. Декораторы 504

Что такое декоратор? 504
Управление вызовами и экземплярами 505
Управление функциями и классами 505

Использование и определение декораторов 506
Для чего используются декораторы? 506
Основы 508
Декораторы функций 508
Декораторы классов 512
Вложение декораторов 514
Аргументы декораторов 516
Декораторы одновременно управляют функциями и классами 517
Реализация декораторов функций 518
Отслеживание вызовов 518
Варианты предохранения состояния для декораторов 519
Грубые ошибки, связанные с классами, часть I: декорирование методов 524
Измерение времени вызовов 530
Добавление аргументов к декоратору 533
Реализация декораторов классов 536
Классы-одиночки 536
Отслеживание объектных интерфейсов 538
Грубые ошибки, связанные с классами, часть II: предохранение множества экземпляров 542
Декораторы или управляющие функции 543
Для чего используются декораторы? (Еще раз) 545
Управление функциями и классами напрямую 547
Пример: “закрытые” и “открытые” атрибуты 549
Реализация закрытых атрибутов 549
Детали реализации, часть I 551
Обобщение также для открытых объявлений 553
Детали реализации, часть II 555
Нерешенные проблемы 556
Python не поощряет контроль доступа 564
Пример: проверка допустимости аргументов функций 565
Цель 565
Базовый декоратор проверки вхождения значений в диапазон для позиционных аргументов 566
Обобщение для поддержки также ключевых аргументов и стандартных значений 568
Детали реализации 571
Нерешенные проблемы 574
Аргументы декоратора или аннотации функций 576
Другие приложения: проверка типов (если вы настаиваете!) 578
Резюме 579
Проверьте свои знания: контрольные вопросы 579
Проверьте свои знания: ответы 580

ГЛАВА 40. Метаклассы 590

Нужно ли иметь дело с метаклассами? 591
Повышение уровней “магии” 592
Язык привязок 593
Недостаток “вспомогательных” функций 594
Метаклассы против декораторов классов: раунд 1 596

Модель метаклассов 599
Классы являются экземплярами `type` 599
Метаклассы являются подклассами `type` 601
Протокол оператора `class` 602
Объявление метаклассов 603
Объявление в Python 3.X 603
Объявление в Python 2.X 604
Координирование метаклассов в Python 3.X и 2.X 605
Реализация метаклассов 605
Базовый метакласс 606
Настройка создания и инициализации 607
Другие методики реализации метаклассов 608
Наследование и экземпляр 613
Метакласс или суперкласс 615
Наследование: вся история 617
Методы метаклассов 623
Методы метаклассов или методы классов 624
Перегрузка операций в методах метакласса 624
Пример: добавление методов в классы 626
Ручное дополнение 626
Дополнение на основе метаклассов 628
Метаклассы против декораторов классов: раунд 2 629
Пример: применение декораторов к методам 634
Трассировка с помощью декорирования вручную 635
Трассировка с помощью метаклассов и декораторов 636
Применение любого декоратора к методам 637
Метаклассы против декораторов классов: раунд 3 (и последний) 639
Резюме 641
Проверьте свои знания: контрольные вопросы 642
Проверьте свои знания: ответы 642

ГЛАВА 41. Все хорошее когда-нибудь заканчивается 644

Парадокс Python 644
О “необязательных” языковых средствах 645
Против тревожных усовершенствований 646
Сложность или мощь 647
Простота или элитарность 648
Заключительные размышления 648
Куда двигаться дальше? 649
На бис: распечатайте собственный сертификат об окончании! 649

Часть IX. Приложения 653

Приложение А. Установка и конфигурирование 654

Установка интерпретатора Python 654
Присутствует ли Python на компьютере? 654
Где взять интерпретатор Python 655
Шаги установки 656

Конфигурирование интерпретатора Python 658
Переменные среды Python 658
Способы установки конфигурационных параметров 660
Аргументы командной строки интерпретатора Python 663
Командные строки запускающего модуля, появившегося в Python 3.3 666
Дополнительная помощь 667

Приложение Б. Запускающий модуль Windows для Python 668

Наследие Unix 668
Наследие Windows 669
Введение в запускающий модуль Windows 670
Учебное руководство по запускающему модулю Windows 672
Шаг 1: использование директив версий в файлах 672
Шаг 2: использование переключателей версий командной строки 675
Выводы: чистый выигрыш для Windows 676

Приложение В. Изменения в Python и настоящая книга 677

Главные отличия между Python 2.X и Python 3.X 677
Отличия Python 3.X 678
Расширения, доступные только в Python 3.X 679
Общие замечания: изменения в Python 3.X 680
Изменения в библиотеках и инструментах 681
Переход на Python 3.X 682
Изменения в Python, относящиеся к пятому изданию: Python 2.7, 3.2, 3.3 682
Изменения в Python 2.7 683
Изменения в Python 3.8 683
Изменения в Python 3.7 683
Изменения в Python 3.3 685
Изменения в Python 3.2 686
Изменения в Python, относящиеся к четвертому изданию: Python 2.6, 3.0, 3.1 686
Изменения в Python 3.1 686
Изменения в Python 3.0 и 2.6 687
Удаления в языке Python 3.0 688
Изменения в Python, относящиеся к третьему изданию: Python 2.3, 2.4, 2.5 691
Более ранние и более поздние изменения в Python 691

Приложение Г. Решения упражнений, приводимых в конце частей 692

Часть VI, “Классы и объектно-ориентированное программирование” 692
Часть VII, “Исключения и инструменты” 700

Предметный указатель 709

Предисловие

По причине большого объема книга разделена на два тома.

Часть I (том 1)

Мы начнем с общего обзора Python, который ответит на часто задаваемые вопросы — почему люди используют язык, для чего он полезен и т.д. В первой главе представлены главные идеи, лежащие в основе технологии, чтобы ввести вас в курс дела. В остальных главах этой части исследуются способы, которыми Python и программисты запускают программы. Главная цель — дать вам достаточный объем информации, чтобы вы были в состоянии работать с последующими примерами и упражнениями.

Часть II (том 1)

Далее мы начинаем тур по языку Python с исследования основных встроенных объектных типов Python и выяснения, что посредством них можно предпринимать: чисел, списков, словарей и т.д. С помощью только этих инструментов уже можно многое сделать, и они лежат в основе каждого сценария Python. Данная часть книги является самой важной, поскольку она образует фундамент для материала, рассматриваемого в оставшихся главах. Здесь мы также исследуем динамическую типизацию и ссылки — ключевые аспекты для правильного применения Python.

Часть III (том 1)

В этой части будут представлены *операторы* Python — код, набираемый для создания и обработки объектов в Python. Здесь также будет описана общая синтаксическая модель Python. Хотя часть сконцентрирована на синтаксисе, в ней затрагиваются связанные инструменты (такие как система PyDoc), концепции итерации и альтернативные способы написания кода.

Часть IV (том 1)

В этой части начинается рассмотрение высокоуровневых инструментов структурирования программ на Python. *Функции* предлагают простой способ упаковки кода для многократного использования и избегания избыточности кода. Здесь мы исследуем правила поиска в областях видимости, приемы передачи аргументов, пресловутые лямбда-функции и многое другое. Мы также пересмотрим итераторы с точки зрения функционального программирования, представим определяемые пользователем генераторы и выясним, как измерять время выполнения кода Python для оценки производительности.

Часть V (том 1)

Модули Python позволяют организовывать операторы и функции в более крупные компоненты; в этой части объясняется, каким образом создавать, применять и перезагружать модули. Мы также обсудим такие темы, как пакеты модулей, перезагрузка модулей, импортирование пакетов, появившиеся в Python 3.3 пакеты пространств имен и атрибут `__name__`.

Часть VI (том 2)

Здесь мы исследуем инструмент объектно-ориентированного программирования Python — *класс*, который является необязательным, но мощным способом структурирования кода для настройки и многократного использования, что почти естественно минимизирует избыточность. Как вы увидите, классы задействуют идеи, раскрытые к этому месту в книге, и объектно-ориентированное программирование в Python сводится главным образом к поиску имен в связанных объектах с помощью специального первого аргумента в функциях. Вы также увидите, что объектно-ориентированное программирование в Python необязательно, но большинство находит объектно-ориентированное программирование на Python более простым, чем на других языках, и оно способно значительно сократить время разработки, особенно при выполнении долгосрочных стратегических проектов.

Часть VII (том 2)

Мы завершим рассмотрение основ языка в книге исследованием модели и операторов обработки исключений Python, а также кратким обзором инструментов разработки, которые станут более полезными, когда вы начнете писать крупные программы (например, инструменты для отладки и тестирования). Хотя исключения являются довольно легковесным инструментом, эта часть помещена после обсуждения классов, поскольку теперь все определяемые пользователем исключения должны быть классами. Мы также здесь раскроем более сложные темы, такие как диспетчеры контекста.

Часть VIII (том 2)

В этой части мы рассмотрим ряд дополнительных тем: Unicode и байтовые строки, инструменты управляемых атрибутов вроде свойств и дескрипторов, декораторы функций и классов и метаклассы. Главы данной части предназначены для дополнительного чтения, т.к. не всем программистам обязательно понимать раскрываемые в них темы. С другой стороны, читатели, которые должны обрабатывать интернационализированный текст либо двоичные данные или отвечать за разработку API-интерфейсов для использования другими программистами, наверняка найдут в этой части что-то интересное для себя. Приводимые здесь примеры крупнее большинства других примеров в книге и могут служить материалом для самостоятельного изучения.

Часть IX (том 2)

Книга завершается четырьмя приложениями, в которых приведены советы по установке и применению Python на разнообразных платформах; представлен запускающий модуль Windows, появившийся в Python 3.3; подытожены изменения, внесенные в различные версии Python; и предложены решения упражнений для каждой части. Ответы на контрольные вопросы по главам приводятся в конце самих глав.

ЧАСТЬ VI

Классы и объектно-ориентированное программирование

ГЛАВА 26

Объектно-ориентированное программирование: общая картина

До сих пор в книге мы использовали термин "объект" в общем смысле. На самом деле код, написанный вплоть до этого момента, был *основанным на объектах* — мы передавали объекты повсюду в сценариях, применяли их в выражениях, вызывали методы объектов и т.д. Однако чтобы код получил право называться подлинно *объектно-ориентированным*, наши объекты, как правило, должны также принимать участие в том, что называется *иерархией наследования*.

В настоящей главе начинается исследование *класса* Python — кодовой структуры и механизма, используемого для реализации в Python новых видов объектов, которые поддерживают наследование. Классы являются главным инструментом объектно-ориентированного программирования (ООП) на языке Python, так что в этой части книги мы также рассмотрим его основы. ООП предлагает отличающийся и часто более эффективный способ программирования, который предусматривает разложение кода на составляющие с целью минимизации избыточности и написания новых программ путем *настройки* существующего кода, а не его изменения на месте.

Классы в Python создаются посредством нового оператора `class`. Как вы увидите, определяемые с помощью классов объекты могут выглядеть очень похожими на встроенные типы, которые мы изучали ранее в книге. В действительности классы всего лишь применяют и расширяют уже раскрытые нами идеи; грубо говоря, они представляют собой пакеты функций, которые используют и обрабатывают объекты встроенных типов. Тем не менее, классы предназначены для создания и управления новыми объектами и поддерживают *наследование* — механизм настройки и многократного применения кода, выходящий за рамки всего того, что мы видели до сих пор.

Одно предварительное замечание: ООП в Python является совершенно необязательным и вам не нужно использовать классы, когда вы только начинаете программировать. Большой объем работы вы можете делать с применением простых конструкций вроде функций и даже кода верхнего уровня сценариев. Поскольку эффективное использование классов требует заблаговременного планирования, они более интересны тем, кто работает в *стратегическом* режиме (занимается долгосрочной разработкой продуктов), нежели тем, кто работает в *тактическом* режиме (испытывая острый дефицит времени).

И все же, как вы увидите в этой части книги, классы оказываются одним из самых полезных инструментов, предоставляемых Python. При надлежащем применении классы способны радикально сократить время разработки. Они также задействованы в популярных инструментах Python наподобие API-интерфейса tkinter, предназначенного для построения графических пользовательских интерфейсов, поэтому большинство программистов на Python обычно сочтут полезным, по крайней мере, практическое знание основ классов.

Для чего используются классы?

Помните ли вы высказывание о том, что программы "делают дела с помощью оснащения", приведенное в главах 4 и 10? Выражаясь простыми терминами, классы являются лишь способом определения новых видов *оснащения*, отражающих реальные объекты в предметной области программы. Например, пусть мы решили реализовать гипотетический робот по приготовлению пиццы, который применялся в качестве примера в главе 16. Если мы реализуем его с использованием классов, то сможем моделировать больше элементов его реальной структуры и взаимосвязей. Здесь полезны следующие два аспекта ООП.

Наследование

Роботы по приготовлению пиццы представляют собой разновидность роботов, поэтому они обладают обычными свойствами, присущими роботам. В терминах ООП мы говорим, что они "наследуют" свойства у общей категории всех роботов. Такие обычные свойства должны быть реализованы только один раз для общего случая и могут многократно применяться частично или полностью всеми типами роботов, которые возможно придется строить в будущем.

Композиция

Роботы по приготовлению пиццы являются совокупностями компонентов, которые работают вместе как одна команда. Скажем, чтобы наш робот был успешным, ему могут понадобиться манипуляторы для раскатывания теста, двигатели для перемещения к духовому шкафу и т.п. Выражаясь в манере ООП, наш робот представляет собой пример композиции; он содержит другие объекты и активизирует их для выполнения соответствующих распоряжений. Каждый компонент может быть реализован в виде класса, который определяет собственное поведение и взаимосвязи.

Общие идеи ООП вроде наследования и композиции применимы к любому приложению, которое может быть разложено на набор объектов. Например, в типовых системах с графическими пользовательскими интерфейсами сами интерфейсы реализованы как совокупности виджетов (кнопок, меток и т.д.), которые рисуются тогда, когда рисуется их контейнер (*композиция*). Кроме того, мы можем располагать возможностью написания собственных виджетов (кнопок с уникальными шрифтами, меток с новыми цветовыми схемами и т.п.), которые являются специализированными версиями общих механизмов интерфейса (*наследование*).

С более конкретной точки зрения программирования классы представляют собой программные единицы Python в точности как функции и модули: они являются еще одним средством для пакетирования логики и данных. На самом деле во многом подобно модулям классы также определяют новые пространства имен. Но по сравнению с другими программными единицами, встречавшимися ранее, классы обладают тремя

важными отличиями, которые делают их более удобными, когда наступает время построения новых объектов.

Множество экземпляров

Классы по существу представляют собой фабрики для генерирования одного и более объектов. При каждом обращении к классу мы генерируем новый объект с отдельным пространством имен. Каждый объект, сгенерированный из класса, имеет доступ к атрибутам класса *и* получает собственное пространство имен для данных, которые варьируются от объекта к объекту. Методика похожа на сохранение состояния для каждого вызова функциями замыканий из главы 17, но в классах она явная и естественная, к тому же отражает лишь одно из дел, обеспечиваемых классами. Классы предлагают завершенное программное решение.

Настройка через наследование

Классы также поддерживают понятие наследования, принятое в ООП; мы можем расширить класс за счет переопределения его атрибутов вне самого класса в новых программных компонентах, реализованных как подклассы. В более общем смысле классы могут образовывать иерархии пространств имен, которые определяют имена для использования объектами, созданными из классов в иерархии. Таким образом, классы поддерживают множество настраиваемых линий поведения более прямо, нежели другие инструменты.

Перегрузка операций

За счет предоставления специальных методов протокола классы могут определять объекты, реагирующие на всевозможные операции, которые мы видели в работе со встроенными типами. Скажем, к созданным с помощью классов объектам можно применять нарезание, конкатенацию, индексирование и т.д. Python предлагает привязки, которые классы могут использовать для перехвата и реализации любой операции для встроенных типов.

По своей сути механизм ООП в Python — это всего лишь *две порции магии*: особый первый аргумент в функциях (для получения объекта, на котором произведен вызов) и поиск в иерархии наследования (для поддержки программирования через настройку). Помимо указанных особенностей модель почти полностью сводится к функциям, которые в конечном итоге обрабатывают встроенные типы. Не являясь радикально новым, ООП добавляет дополнительный уровень структуры, которая поддерживает более эффективное программирование, чем обычные процедурные модели. Наряду с рассмотренными ранее функциональными инструментами ООП олицетворяет собой значительный шаг в сторону абстрагирования от компьютерного оборудования, который помогает нам строить более сложно устроенные программы.

Объектно-ориентированное программирование с высоты птичьего полета

Прежде чем мы увидим, что все это означает в переводе на код, я хотел бы кратко коснуться общих идей, лежащих в основе ООП. Если до сих пор вы не делали ничего объектно-ориентированного, тогда некоторые термины в настоящей главе на первый взгляд могут показаться слегка озадачивающими. Более того, мотивировка данных терминов может ускользать, пока вы не получите возможность изучить способы,

которыми программисты применяют их в более крупных системах. ООП – такая же практика, как и технология.

Поиск в иерархии наследования

Хорошая новость в том, что ООП в Python гораздо проще для понимания и использования, чем в других языках, таких как C++ или Java. Будучи динамически типизированным языком написания сценариев, Python устраняет большую часть синтаксического беспорядка и сложности, которые затуманивают ООП в других инструментах. На самом деле большинство истории ООП в Python сводится до следующего выражения:

```
объект.атрибут
```

Мы применяли такое выражение повсюду в книге для доступа к атрибутам модуля, вызова методов объектов и т.п. Однако когда мы используем его в отношении объекта, который получен из оператора `class`, выражение инициирует *поиск* в Python – он ищет в дереве связанных объектов первое появление `атрибута`. Вот что фактически предусматривает предыдущее выражение Python при участии классов:

Найти первое вхождение `атрибута`, просматривая `объект`, а затем все классы выше него, снизу вверх и слева направо.

Другими словами, извлечение атрибутов – это просто поиск в дереве. Термин *наследование* применяется из-за того, что объекты ниже в дереве наследуют атрибуты, присоединенные к объектам выше в дереве. По мере того, как поиск продолжается снизу вверх, связанные в дерево объекты в некотором смысле представляют собой объединение всех атрибутов, определенных во всех родителях в дереве на всем пути вверх.

В Python все происходит буквально: мы действительно строим дерево связанных объектов с помощью кода, а Python во время выполнения на самом деле поднимается по такому дереву в поисках атрибутов каждый раз, когда встречается выражение `объект.атрибут`. Пример одного из таких деревьев показан на рис. 26.1.

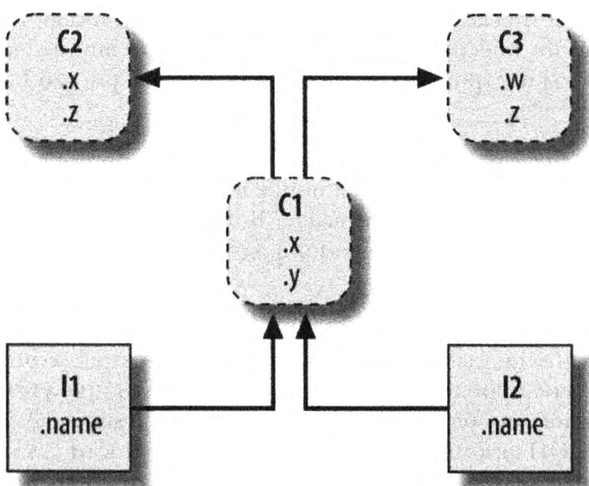


Рис. 26.1. *Дерево классов с двумя экземплярами в нижней части (`I1` и `I2`), классом выше них (`C1`) и двумя суперклассами в верхней части (`C2` и `C3`). Все эти объекты являются пространствами имен (пакетами переменных), и поиск в иерархии наследования представляет собой просто обход дерева снизу вверх с целью нахождения самого нижнего вхождения атрибута. Код заключает в себе форму таких деревьев*

На рис. 26.1 изображено дерево из пяти объектов, помеченных переменными, которые имеют присоединенные атрибуты, готовые для поиска. Более конкретно дерево связывает вместе три *объекта классов* (овалы C1, C2 и C3) и два *объекта экземпляров* (прямоугольники I1 и I2), образуя дерево поиска в иерархии наследования. Обратите внимание, что в объектной модели Python классы и генерируемые из них экземпляры являются двумя отдельными типами объектов.

Классы

Служат фабриками экземпляров. Атрибуты классов обеспечивают поведение (данные и функции), которое наследуется всеми экземплярами, сгенерированными из них (например, функция для расчета заработной платы сотрудника на основе оклада и отработанных часов).

Экземпляры

Представляют конкретные элементы в предметной области программы. Атрибуты экземпляров хранят данные, которые варьируются для каждого отдельного объекта (скажем, номер карточки социального страхования сотрудника).

С точки зрения деревьев поиска экземпляр наследует атрибуты от своего класса, а класс наследует атрибуты от всех классов выше в дереве.

На рис. 26.1 мы можем продолжить категоризацию овалов по их относительным позициям в дереве. Классы, расположенные более высоко в дереве (наподобие C2 и C3), мы обычно называем *суперклассами*; классы, находящиеся ниже в дереве (вроде C1) известны как *подклассы*. Указанные термины касаются относительных позиций в дереве и исполняемых ролей. Суперклассы обеспечивают поведение, разделяемое всеми их подклассами, но поскольку поиск направлен снизу вверх, подклассы могут переопределять поведение, определенное их суперклассами, за счет переопределения имен суперклассов ниже в дереве[1].

Так как последние несколько слов на самом деле отражают суть работы по настройке программного обеспечения в ООП, давайте расширим данную концепцию. Предположим, что мы построили дерево, приведенное на рис. 26.1, и затем написали:

```
I2.w
```

Код сразу же обращается к наследованию. Поскольку он представляет собой выражение *объект.атрибут*, инициируется поиск в дереве, изображенном на рис. 26.1 — Python будет искать атрибут w в I2 и выше. В частности, он будет проводить поиск внутри связанных объектов в следующем порядке:

```
I2, C1, C2, C3
```

и остановится при нахождении первого присоединенного атрибута w (или сообщит об ошибке, если w не удалось отыскать). В данном случае атрибут w не будет найден до тех пор, пока не пройдет поиск в C3, потому что он присутствует только в этом объекте. Другими словами, благодаря автоматическому поиску I2.w распознается как C3.w. В терминах ООП экземпляр I2 "наследует" атрибут w от C3.

В конечном итоге два экземпляра наследуют от своих классов четыре атрибута: w, x, y и z. Ссылки на другие атрибуты будут вызывать проход по другим путям в дереве.

[1] В других книгах и группах людей можно также встретить термины *базовые классы* и *производные классы*, применяемые для обозначения суперклассов и подклассов. Программисты на Python предпочитают использовать термины *суперклассы* и *подклассы*.

Вот примеры.

- Для `I1.x` и `I2.x` атрибут `x` обнаруживается в `C1` и поиск останавливается, т.к. `C1` располагается в дереве ниже, чем `C2`.
- Для `I1.y` и `I2.y` атрибут `y` обнаруживается в `C1`, поскольку это единственное место, где присутствует `y`.
- Для `I1.z` и `I2.z` атрибут `z` обнаруживается в `C2`, потому что `C2` располагается в дереве левее, чем `C3`.
- Для `I2.name` атрибут `name` обнаруживается в `I2` вообще без подъема по дереву.

Отследите описанные варианты в дереве на рис. 26.1, чтобы получить представление о том, как работает поиск в иерархии наследования в Python.

В предыдущем списке первый элемент является, вероятно, наиболее важным — поскольку класс `C1` переопределяет атрибут `x` ниже в дереве, он фактически *замещает* его версию, находящуюся выше в `C2`. Как вы вскоре увидите, такие переопределения являются центральной частью настройки программного обеспечения в ООП — за счет переопределения и замещения атрибута класс `C1` по существу настраивает то, что он наследует от своих суперклассов.

Классы и экземпляры

Хотя формально классы и экземпляры, помещаемые в деревья наследования, являются разными типами объектов в модели Python, они практически идентичны; главная цель каждого типа заключается в том, чтобы служить еще одним видом *пространства имен* — пакета переменных и места, куда мы можем присоединять атрибуты. Следовательно, если классы и экземпляры выглядят подобными модулям, то так и должно быть; тем не менее, объекты в деревьях классов также имеют автоматически просматриваемые ссылки на другие объекты пространств имен и классы соответствуют операторам, а не целым файлам.

Основное отличие между классами и экземплярами состоит в том, что классы представляют собой своего рода *фабрики* для генерирования экземпляров. Скажем, в реалистичном приложении мы могли бы иметь класс `Employee`, который определяет все то, что характерно для сотрудника; из этого класса мы генерируем действительные экземпляры `Employee`. Есть еще одно отличие между классами и модулями — мы можем иметь только один экземпляр отдельного модуля в памяти (именно потому модуль приходится перезагружать, чтобы получить его новый код), но для классов допускается создавать столько экземпляров, сколько нужно.

Что касается эксплуатации, то классы обычно будут располагать присоединенными к ним функциями (например, `computeSalary`), а экземпляры будут иметь больше базовых элементов данных, используемых функциями класса (скажем, `hoursWorked`). На самом деле объектно-ориентированная модель ничем не отличается от классической модели обработки данных с *программами* и *записями* — в ООП экземпляры похожи на записи с "данными", а классы являются "программами" для обработки этих записей. Однако в ООП также присутствует понятие иерархии наследования, которая лучше поддерживает настройку программного обеспечения, чем более ранние модели.

Вызовы методов

В предыдущем разделе мы видели, что ссылка на атрибут `I2.w` в примере дерева классов была оттранслирована в `C3.w` посредством процедуры поиска внутри иерархии наследования в Python. Тем не менее, столь же важно понимать, что происходит,

когда мы пытаемся вызывать *методы* — функции, присоединенные к классам в качестве атрибутов.

Если ссылка `I2.w` представляет собой вызов *функции*, тогда в действительности она означает "вызвать функцию `C3.w` для обработки `I2`". То есть Python будет автоматически отображать вызов `I2.w()` на вызов `C3.w(I2)`, передавая унаследованной функции экземпляр в первом аргументе.

Фактически всякий раз, когда вызывается функция, подобным образом присоединенная к классу, всегда подразумевается экземпляр класса. Подразумеваемый объект или контекст отчасти является причиной того, что мы называем это *объектно-ориентированной* моделью — при выполнении операции всегда имеется подчиненный объект. В более реалистичном примере мы могли бы вызывать метод повышения по имени `giveRaise`, присоединенный в виде атрибута к классу сотрудника `Employee`; такой вызов не имеет смысла, если только он не уточнен объектом сотрудника, в отношении которого должно быть применено повышение.

Как мы увидим позже, Python передает подразумеваемый экземпляр методу в особом первом аргументе, который по соглашению называется `self`. Методы принимают этот аргумент для обработки объекта, на котором произведен вызов. Как мы также узнаем, методы можно вызывать либо через экземпляр (`bob.giveRaise()`), либо через класс (`Employee.giveRaise(bob)`), и обе формы служат своим целям в наших сценариях. Вызовы также иллюстрируют обе ключевые идеи в ООП: ниже описано, что делает Python, чтобы выполнить вызов метода `bob.giveRaise()`.

1. Ищет `giveRaise` в `bob` с помощью поиска в иерархии наследования.
2. Передает `bob` найденной функции `giveRaise` в особом аргументе `self`.

Когда вы вызываете `Employee.giveRaise(bob)`, то всего лишь самостоятельно выполняете оба шага. Формально приведенное описание отражает стандартный случай (в Python имеются дополнительные типы методов, с которыми мы встретимся позже), но оно применимо к подавляющему большинству написанного кода с использованием ООП. Однако чтобы посмотреть, как методы принимают свои подчиненные объекты, нам необходимо перейти к написанию какого-нибудь кода.

Создание деревьев классов

Несмотря на то что мы говорим обо всех идеях абстрактно, конечно же, за ними стоит овеществленный код. Мы создадим с помощью операторов `class` и обращений к классам деревья и их объекты, которые затем исследуем более детально. Ниже приведена краткая сводка.

- Каждый оператор `class` генерирует новый объект класса.
- При каждом обращении к классу он генерирует новый объект экземпляра.
- Экземпляры автоматически связываются с классами, из которых они были созданы.
- Классы автоматически связываются со своими суперклассами в соответствии со способом их перечисления внутри круглых скобок в строке заголовка `class`; порядок слева направо здесь дает порядок в дереве.

Например, чтобы построить дерево, показанное на рис. 26.1, мы могли бы запустить представленный далее код Python. Подобно определениям функций код классов обычно помещается в файлы модулей и выполняется во время импортирования (для краткости внутренности операторов `class` опущены):

```
class C2: ...          # Создание объектов классов (овалов)
class C3: ...
class C1(C2, C3): ... # Связывание с его суперклассами (в указанном порядке)

I1 = C1()              # Создание объектов экземпляров (прямоугольников)
I2 = C1()              # Связывание с его классом
```

Здесь мы создаем три объекта классов посредством трех операторов class и два объекта экземпляров за счет двукратного обращения к классу C1, как если бы он был функцией. Экземпляры запоминают класс, из которого были созданы, а класс C1 запоминает свои перечисленные суперклассы.

Формально в примере применяется то, что называется *множественным наследованием*, которое просто означает наличие у класса более одного суперкласса выше в дереве классов — удобный прием, когда желательно объединить несколько инструментов. В Python при перечислении более одного суперкласса внутри круглых скобок в операторе class (как выше в C1) их порядок слева направо задает порядок, в котором будет производиться поиск атрибутов в этих суперклассах. По умолчанию используется крайняя слева версия имени, хотя вы всегда можете выбрать имя, запросив его у класса, где имя находится (скажем, C3.z).

Из-за особенностей выполнения поиска в иерархии наследования объект, к которому вы присоединяете атрибут, оказывается критически важным — он определяет область видимости имени. Атрибуты, присоединенные к экземплярам, сохраняются только для этих одиночных экземпляров, но атрибуты, присоединенные к классам, разделяются всеми их подклассами и экземплярами. Позже мы более глубоко исследуем код, который присоединяет атрибуты к таким объектам. Мы увидим, что:

- атрибуты обычно присоединяются к классам с помощью присваиваний, выполняемых на верхнем уровне блоков операторов class, а не во вложенных операторах def определения функций;
- атрибуты обычно присоединяются к экземплярам посредством присваиваний особому аргументу, передаваемому функциям внутри классов, по имени self.

Например, классы обеспечивают поведение для своих экземпляров с помощью функций методов, которые мы создаем за счет написания кода операторов def внутри операторов class. Поскольку такие вложенные операторы def присваивают имена внутри класса, они в итоге присоединяют к объекту класса атрибуты, которые будут наследоваться всеми экземплярами и подклассами:

```
class C2: ...                    # Создание объектов суперклассов
class C3: ...

class C1(C2, C3):                # Создание и связывание класса C1
    def setname(self, who):      # Присваивание name: C1.setname
        self.name = who          # self является либо I1, либо I2

I1 = C1()                        # Создание двух экземпляров
I2 = C1()
I1.setname('bob')                # Установка I1.name в 'bob'
I2.setname('sue')                # Установка I2.name в 'sue'
print(I1.name)                   # Выводит 'bob'
```

Ничего уникального в плане синтаксиса def в этом контексте нет. Тем не менее, с точки зрения эксплуатации, когда оператор def появляется внутри class, он обычно

известен как *метод* и автоматически получает особый первый аргумент, по соглашению называемый self, который предоставляет возможность обращаться к обрабатываемому экземпляру. Любые значения, которые вы передаете методу самостоятельно, отправляются аргументам, следующим после self (здесь who)[2].

Из-за того, что классы представляют собой фабрики для множества экземпляров, их методы обычно задействуют этот автоматически передаваемый аргумент self всякий раз, когда необходимо извлекать или устанавливать атрибуты отдельного экземпляра, обрабатываемого вызовом метода. В предыдущем коде self применяется для сохранения name в одном из двух экземпляров.

Как и простые переменные, атрибуты классов и экземпляров не объявляются заблаговременно, но появляются во время присваивания значений в первый раз. Когда метод выполняет присваивание атрибуту self, он создает или изменяет атрибут в экземпляре в нижней части дерева классов (т.е. один из прямоугольников на рис. 26.1), потому что self автоматически ссылается на обрабатываемый экземпляр — объект, на котором произведен вызов.

На самом деле, поскольку все объекты в деревьях классов являются всего лишь объектами пространств имен, мы можем извлекать или устанавливать любой из их атрибутов, указывая подходящее имя. Выражение C1.setname в той же мере допустимо, как и I1.setname, при условии, что имена C1 и I1 находятся в области видимости вашего кода.

Перегрузка операций

В текущем виде наш класс C1 не присоединяет атрибут name к экземпляру до тех пор, пока не будет вызван метод setname. В действительности ссылка I1.name перед вызовом I1.setname привела бы к возникновению ошибки неопределенного имени. Если в классе желательно гарантировать, что атрибут вроде name всегда устанавливается в экземплярах, тогда более типично будет заполнять его во время конструирования:

```
class C2: ...                    # Создание объектов суперклассов
class C3: ...

class C1(C2, C3):
    def __init__(self, who):     # Установка name при конструировании
        self.name = who          # self является либо I1, либо I2

I1 = C1('bob')                   # Установка I1.name в 'bob'
I2 = C1('sue')                   # Установка I2.name в 'sue'
print(I1.name)                   # Выводит 'bob'
```

Каждый раз, когда экземпляр генерируется из класса, Python автоматически вызывает метод по имени __init__, будь он реализован или унаследован. Как обычно, новый экземпляр передается в аргументе self метода __init__, а значения, перечисленные в круглых скобках при обращении к классу, передаются второму и последующим аргументам. Результатом оказывается инициализация экземпляров, когда они создаются, без необходимости в дополнительных вызовах методов.

Метод __init__ известен как *конструктор* из-за момента своего запуска. Он является самым часто используемым представителем крупной группы методов, называе-

[2] Если вам когда-либо приходилось использовать язык C++ или Java, то вы заметите, что self в Python — то же самое, что и указатель this, но self всегда задается явно в заголовках и телах методов Python, чтобы сделать доступ к атрибутам более очевидным: имя имеет меньше возможных предназначений.

мых *методами перегрузки операций*, которые мы обсудим более подробно в последующих главах. Такие методы обычным образом наследуются в деревьях классов и содержат в начале и конце своих имен по два символа подчеркивания, чтобы акцентировать внимание на их особенности. Python запускает их автоматически, когда экземпляры, поддерживающие методы, встречаются в соответствующих операциях, и они главным образом выступают в качестве альтернативы применению простых вызовов методов. Кроме того, они необязательны: если операции опущены, то они не поддерживаются. При отсутствии `__init__` обращения к классам возвращают пустые экземпляры без их инициализации.

Скажем, для реализации пересечения множеств класс может либо предоставить метод по имени `intersect`, либо перегрузить операцию выражения `&` и обеспечить требующуюся логику за счет реализации метода по имени `__and__`. Поскольку схема с операциями делает экземпляры больше похожими на встроенные типы, она позволяет ряду классов предоставлять согласованный и естественный интерфейс, а также быть совместимыми с кодом, который ожидает встроенного типа. Однако за исключением конструктора `__init__`, который присутствует в большинстве реалистичных классов, во многих программах лучше использовать более просто именованные методы, если только их объекты не подобны объектам встроенных типов. Метод `giveRaise` может иметь смысл для класса `Employee`, но операция `&` — нет.

Объектно-ориентированное программирование — это многократное использование кода

Наряду с несколькими синтаксическими деталями все описанное ранее является значительной частью истории об ООП в Python. Разумеется, здесь есть нечто большее, нежели просто наследование. Например, перегрузка операций намного более универсальна, чем обсуждалось до сих пор — классы могут также предоставлять собственные реализации операций, таких как индексирование, извлечение атрибутов, вывод и т.д. Тем не менее, в общем ООП сводится к поиску атрибутов в деревьях и особому первому аргументу в функциях.

Почему нас может интересовать создание и поиск в деревьях объектов? При надлежащем применении классы поддерживают *многократное использование* кода способами, которые не могут обеспечить другие программные компоненты Python, хотя для этого необходим определенный опыт. Фактически в том и заключается их наивысшая цель. С помощью классов мы настраиваем существующее программное обеспечение вместо того, чтобы либо изменять имеющийся код на месте, либо начинать с нуля каждый новый проект. В итоге они оказываются мощной парадигмой в реальном программировании.

На фундаментальном уровне классы являются всего лишь пакетами функций и других имен, что очень похоже на модули. Однако получаемый благодаря классам автоматический поиск атрибутов в иерархии наследования поддерживает настройку программного обеспечения, которая выходит за рамки того, что можно делать посредством модулей и функций. Кроме того, классы обеспечивают естественную структуру для кода, которая упаковывает и локализует логику и имена, что оказывает помощь в отладке.

Например, поскольку методы представляют собой функции с особым первым аргументом, мы можем частично имитировать их поведение, вручную передавая подлежащие обработке объекты простым функциям. Тем не менее, участие методов в наследовании классов позволяет нам естественным образом настраивать существующее

программное обеспечение путем создания подклассов с новыми определениями методов, а не изменять имеющийся код на месте. В случае модулей и функций такой возможности нет.

Полиморфизм и классы

В качестве примера предположим, что вам поручили реализовать приложение с базой данных сотрудников. Как программист на Python, применяющий ООП, вы можете начать с создания универсального суперкласса, в котором определены стандартные линии поведения, общие для всех типов сотрудников в организации:

```
class Employee:                          # Универсальный суперкласс
    def computeSalary(self): ...         # Общие или стандартные линии поведения
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

После написания кода общего поведения вы можете специализировать его для каждого индивидуального типа сотрудника, отражая его отличия от нормы. То есть вы можете создавать подклассы, настраивающие только те фрагменты поведения, которые отличаются в зависимости от типа сотрудника; остальное поведение будет унаследовано от более универсального класса. Скажем, если с инженерами связано уникальное правило подсчета заработной платы (возможно, они не на почасовой оплате), тогда вы можете заменить в подклассе только один метод:

```
class Engineer(Employee):                # Специализированный подкласс
    def computeSalary(self): ...         # Что-то специальное
```

Из-за того, что версия computeSalary находится ниже в дереве классов, она заместит (переопределит) универсальную версию в Employee. Затем вы создаете экземпляры разновидностей классов сотрудников, к которым принадлежат реальные сотрудники, чтобы получить корректное поведение:

```
bob = Employee()    # Стандартное поведение
sue = Employee()    # Стандартное поведение
tom = Engineer()    # Специальный расчет заработной платы
```

Обратите внимание, что вы можете создавать экземпляры любого класса в дереве, а не только классов в нижней части — класс, из которого вы создаете экземпляр, определяет уровень, откуда будет начинаться поиск атрибутов, и соответственно то, какие версии методов он будет задействовать.

В конце концов, эти три объекта экземпляров могут оказаться встроенными в более крупный контейнерный объект (например, список или экземпляр другого класса), который представляет отдел или компанию, воплощая упомянутую в начале главы идею композиции. Когда вы позже запросите заработные платы сотрудников, они будут рассчитываться в соответствии с классами, из которых создавались объекты, благодаря принципам поиска в иерархии наследования:

```
company = [bob, sue, tom]                # Составной объект
for emp in company:
    print(emp.computeSalary())           # Выполнить версию для данного объекта:
                                         #   стандартную или специальную
```

Мы имеем еще одну иллюстрацию идеи *полиморфизма*, которая была представлена в главе 4 и расширена в главе 16. Как вы наверняка помните, полиморфизм означает, что смысл операции зависит от объекта, с которым она работает. Таким образом, код

не должен заботиться о том, чем объект *является*, а лишь о том, что он *делает*. Перед вызовом метод computeSalary ищется в иерархии наследования для каждого объекта. Совокупный эффект заключается в том, что мы автоматически запускаем корректную версию для обрабатываемого объекта. Для лучшего понимания отследите код[3].

В других приложениях полиморфизм также может использоваться для сокрытия (т.е. *инкапсуляции*) отличий в интерфейсах. Скажем, программа обработки потоков данных может быть реализована так, чтобы ожидать объекты с методами ввода и вывода, не заботясь о том, что в действительности делают эти методы:

```
def processor(reader, converter, writer):
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Передавая экземпляры подклассов, которые специализируют обязательные интерфейсные методы read и write для различных источников данных, мы можем многократно применять функцию processor для любого необходимого источника данных, как сейчас, так и в будущем:

```
class Reader:
    def read(self): ...         # Стандартное поведение и инструменты
    def other(self): ...
class FileReader(Reader):
    def read(self): ...         # Читать из локального файла
class SocketReader(Reader):
    def read(self): ...         # Читать из сетевого сокета
...
processor(FileReader(...), Converter, FileWriter(...))
processor(SocketReader(...), Converter, TapeWriter(...))
processor(FtpReader(...), Converter, XmlWriter(...))
```

Более того, поскольку внутренние реализации методов read и write вынесены в отдельные места, их можно изменять, не оказывая влияние на код, в котором они используются. Функцию processor можно было бы даже превратить в класс, чтобы позволить логике преобразования converter наполняться через наследование и дать возможность подклассам чтения и записи встраиваться посредством композиции (позже в данной части книги будет показано, как это работает).

Программирование путем настройки

Как только вы привыкнете к программированию в таком стиле (путем настройки программного обеспечения), вы обнаружите, что когда наступает время написания новой программы, бо́льшая часть работы может оказаться сделанной — в значительной степени ваша задача сводится к смешиванию существующих суперклассов, которые уже реализуют поведение, требующееся для программы. Например, кто-то другой мог написать классы Employee, Reader и Writer для применения в совершенно разных программах. В таком случае вы получаете весь их код "бесплатно".

---

[3] Список company в приведенном примере мог бы быть базой данных в случае хранения в файле посредством модуля pickle, представленного в главе 9, чтобы обеспечить постоянство для объектов сотрудников. Python также поставляется с модулем по имени shelve, который позволяет сохранять обработанные с помощью pickle представления экземпляров классов в файловой системе с доступом по ключу; мы рассмотрим эту тему в главе 28.

Фактически во многих прикладных областях вы можете выбрать или приобрести наборы суперклассов, известные как *фреймворки*, которые реализуют распространенные задачи программирования в виде классов, готовых к смешиванию в ваших приложениях. Подобные фреймворки могут предоставлять интерфейсы к базам данных, протоколы тестирования, комплекты инструментов для построения графических пользовательских интерфейсов и т.д. Располагая фреймворками, вы часто просто пишете код подкласса, в котором реализуете один или два ожидаемых метода; бо́льшую часть работы выполняют классы фреймворков, находящиеся выше в дереве. Программирование в таком мире ООП представляет собой всего лишь комбинирование и специализацию уже отлаженного кода за счет написания собственных подклассов.

Конечно, обучение тому, как задействовать классы для достижения идеального мира ООП, требует времени. На практике ООП также влечет за собой значительную работу по проектированию, чтобы получить все преимущества от многократного использования кода классов. С этой целью программисты начали каталогизировать распространенные структуры ООП, известные как *паттерны проектирования*, которые призваны помочь в решении проблем, возникающих при проектировании. Однако действительный код, который вы пишете с применением ООП в Python, настолько прост, что сам по себе он не будет дополнительным препятствием для вашего путешествия в ООП. Чтобы удостовериться в этом, читайте главу 27.

Резюме

В главе мы кратко рассмотрели классы и ООП, предоставив общую картину, прежде чем углубляться в детали синтаксиса. Как вы видели, ООП — это в основном аргумент по имени `self` и поиск атрибутов в деревьях связанных объектов, называемых наследованием. Объекты в нижней части дерева наследуют атрибуты от объектов выше в дереве — характеристика, которая делает возможным программирование путем настройки кода, а не его изменения или написания с нуля. При надлежащем использовании такая модель программирования может радикально сократить время разработки.

В следующей главе начнется наполнение общей картины недостающими деталями написания кода. Тем не менее, по мере углубления в классы имейте в виду, что модель ООП в Python очень проста; как будет показано, в действительности она сводится всего лишь к поиску атрибутов в деревьях объектов и особому аргументу функций. Но до того как двигаться дальше, закрепите пройденный материал главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. В чем сущность ООП в Python?
2. Где процедура поиска в иерархии наследования ищет атрибуты?
3. В чем отличие между объектом класса и объектом экземпляра?
4. Почему первый аргумент в функции метода класса является особым?
5. Для чего применяется метод `__init__`?
6. Как бы вы создали экземпляр класса?
7. Как бы вы создали класс?
8. Как бы вы указали суперклассы класса?

Проверьте свои знания: ответы

1. ООП предназначено для многократного использования кода — вы производите разложение кода с целью минимизации избыточности и программируете путем настройки того, что уже существует, а не изменяете код на месте или пишете его с нуля.

2. Процедура поиска в иерархии наследования ищет атрибут сначала в объекте экземпляра, затем в классе, из которого был создан экземпляр, далее во всех расположенных выше суперклассах, двигаясь от нижней части дерева объектов к его верхней части и слева направо (по умолчанию). Поиск останавливается в первом месте, где найден атрибут. Поскольку выигрывает самая нижняя версия имени, найденная в ходе поиска, иерархии классов естественным образом поддерживают настройку путем расширения в новых подклассах.

3. Объекты классов и объекты экземпляров представляют собой пространства имен (пакеты переменных, которые выступают в качестве атрибутов). Основное отличие между ними в том, что классы являются своего рода фабриками для создания множества экземпляров. Классы также поддерживают методы перегрузки операций, которые экземпляры наследуют, а любые функции, вложенные внутрь классов, трактуются как методы для обработки экземпляров.

4. Первый аргумент в функции метода класса является особым, потому что он всегда получает объект экземпляра, представляющий собой подразумеваемый объект, на котором вызван метод. По соглашению он называется `self`. Поскольку по умолчанию функции методов всегда имеют такой подразумеваемый объект и объектный контекст, мы говорим, что они являются "объектно-ориентированными" (т.е. предназначенными для обработки либо изменения объектов).

5. Метод `__init__` реализуется или наследуется в классе, и Python вызывает его автоматически каждый раз, когда создается экземпляр этого класса. Он известен как метод конструктора; ему неявно передается новый экземпляр, а также любые аргументы, указанные явно с именем класса. Кроме того, он является наиболее часто применяемым методом перегрузки операций. В случае отсутствия метода `__init__` экземпляры просто начинают свое существование как пустые пространства имен.

6. Вы создаете экземпляр класса с помощью обращения к имени класса так, как если бы оно было функцией. Любые аргументы, указанные с именем класса, становятся вторым и последующими аргументами в методе конструктора `__init__`. Новый экземпляр запоминает класс, из которого он был создан, для целей, связанных с наследованием.

7. Вы создаете класс посредством выполнения оператора `class`; подобно определениям функций такие операторы обычно выполняются при импортировании включающего модуля (более подробно об этом речь пойдет в следующей главе).

8. Вы указываете суперклассы класса, перечисляя их внутри круглых скобок в операторе `class` после имени нового класса. Порядок слева направо, в котором классы перечисляются в круглых скобках, дает порядок слева направо при поиске в иерархии наследования, представленной деревом классов.

ГЛАВА 27

Основы написания классов

Теперь, когда был представлен краткий обзор ООП, самое время посмотреть, каким образом все выливается в фактический код. В этой главе начинается наполнение недостающими синтаксическими деталями модели классов в Python.

Если в прошлом вы не занимались ООП, тогда классы поначалу могут показаться отчасти сложными. Чтобы облегчить освоение программирования классов, мы начнем подробное исследование ООП с того, что рассмотрим в настоящей главе несколько базовых классов в действии. Приведенные здесь детали будут расширены в последующих главах части, но в своей элементарной форме классы Python понимать легко.

На самом деле классы обладают только тремя отличительными особенностями. На базовом уровне они главным образом представляют собой пространства имен, что во многом подобно модулям, которые обсуждались в части V. Однако в отличие от модулей классы также поддерживают генерирование множества объектов, наследование пространств имен и перегрузку операций. Давайте начнем наш тур по оператору `class` с исследования каждой из этих отличительных особенностей по очереди.

Классы генерируют множество объектов экземпляров

Чтобы понять, как работает идея множества объектов, сначала необходимо осознать, что в модели ООП языка Python имеются два вида объектов: объекты *классов* и объекты *экземпляров*. Объекты классов обеспечивают стандартное поведение и служат фабриками для объектов экземпляров. Объекты экземпляров являются действительными объектами, обрабатываемыми вашей программой — каждый представляет собой самостоятельное пространство имен, но наследует (т.е. автоматически получает доступ) имена от класса, из которого он был создан. Объекты классов происходят из операторов, а экземпляры — из вызовов; при каждом обращении к классу вы получаете новый экземпляр этого класса.

Такая концепция генерации объектов сильно отличается от большинства других программных конструкций, рассмотренных до сих пор в книге. По существу классы являются *фабриками* для генерирования множества экземпляров. По контрасту с этим в отдельно взятой программе импортируется только одна копия каждого модуля. Фактически именно потому так работает функция `reload`, обновляя разделяемый объект единственного экземпляра на месте. Благодаря классам каждый экземпляр мо-

жет иметь собственные независимые данные, поддерживая множество версий объекта, который моделируется классом.

В данной роли экземпляры классов похожи на поддерживаемое для каждого вызова состояние *замыканий* (фабричных функций), которые обсуждались в главе 17, но они представляют собой естественную часть модели классов, а состояние в классах реализовано в виде явных атрибутов вместо неявных ссылок на области видимости. Вдобавок это всего лишь часть того, что делают классы — они также поддерживают настройку через наследование, перегрузку операций и множество линий поведения через методы. Вообще говоря, классы являются более совершенным программным инструментом, хотя ООП и *функциональное программирование* не считаются взаимоисключающими парадигмами. Мы можем сочетать их, используя инструменты функционального программирования в методах, реализуя методы, которые сами представляют собой генераторы, создавая определяемые пользователем итераторы (как будет показано в главе 30) и т.д.

Ниже приведен краткий обзор основных возможностей ООП в Python с точки зрения двух типов объектов. Как вы увидите, классы Python в чем-то похожи на определения `def` и модули, но они могут сильно отличаться от того, к чему вы привыкли в других языках.

Объекты классов обеспечивают стандартное поведение

В результате выполнения оператора `class` мы получаем объект класса. Далее представлена сводка по основным характеристикам классов Python.

- Оператор `class` создает объект класса и присваивает его имени. В точности как оператор `def` определения функции оператор `class` является исполняемым. После достижения и запуска он генерирует новый объект класса и присваивает его имени, указанному в заголовке `class`. Также подобно `def` операторы `class` обычно выполняются при первом импортировании файлов, где они находятся.
- Присваивания внутри операторов `class` создают атрибуты классов. Как и в файлах модулей, присваивания на верхнем уровне внутри оператора `class` (не вложенные в `def`) генерируют атрибуты в объекте класса. Формально оператор `class` определяет локальную область видимости, которая превращается в пространство имен атрибутов для объекта класса подобно глобальной области видимости модуля. После выполнения оператора `class` атрибуты класса доступны посредством уточнения с помощью имени: `объект.имя`.
- Атрибуты класса снабжают объект состоянием и поведением. Атрибуты объекта класса хранят информацию о состоянии и описывают поведение, которое разделяется всеми экземплярами, создаваемыми из класса; операторы `def` определения функций, вложенные внутрь `class`, генерируют методы, которые обрабатывают экземпляры.

Объекты экземпляров являются конкретными элементами

При обращении к объекту класса мы получаем объект экземпляра. Ниже приведен краткий обзор ключевых моментов, касающихся экземпляров класса.

- Обращение к объекту класса как к функции создает новый объект экземпляра. При каждом обращении к классу он создает и возвращает новый объект экземпляра. Экземпляры представляют конкретные элементы в предметной области программы.

- Каждый объект экземпляра наследует атрибуты класса и получает собственное пространство имен. Объекты экземпляров, созданные из классов, являются новыми пространствами имен. Объекты экземпляров начинают свое существование пустыми, но наследуют атрибуты, имеющиеся в объектах классов, из которых они были сгенерированы.
- Присваивания атрибутам аргумента `self` в методах создают атрибуты для отдельного экземпляра. Внутри функций методов класса первый аргумент (по соглашению называемый `self`) ссылается на обрабатываемый объект экземпляра; присваивания атрибутам аргумента `self` создают либо изменяют данные в экземпляре, но не в классе.

Конечным результатом оказывается то, что классы определяют общие разделяемые данные и поведение плюс генерируют экземпляры. Экземпляры отражают конкретные сущности приложения и хранят собственные данные, которые могут варьироваться от объекта к объекту.

Первый пример

Давайте рассмотрим реальный пример, чтобы увидеть, как описанные выше идеи работают на практике. Первым делом определим класс по имени `FirstClass`, выполнив оператор `class` в интерактивной подсказке:

```
>>> class FirstClass:                 # Определить объект класса
        def setdata(self, value):     # Определить методы класса
            self.data = value         # self - это экземпляр
        def display(self):
            print(self.data)          # self.data: для каждого экземпляра
```

Здесь мы работаем в интерактивной подсказке, но по обыкновению такой оператор находится в файле модуля и выполняется при его импортировании. Подобно функциям, создаваемым с помощью операторов `def`, этот класс не появится до тех пор, пока Python не доберется и не выполнит показанный оператор.

Как и все составные операторы, оператор `class` начинается со строки заголовка с именем класса, после чего следует тело с одним или несколькими вложенными операторами, (обычно) набранными с отступом. В приведенном примере вложенными операторами являются `def`; они определяют функции, которые реализуют поведение класса, предназначенное для экспортирования.

В части IV было указано, что `def` на самом деле представляет собой присваивание. В примере операторы `def` присваивают объекты функций именам `setdata` и `display` в области видимости оператора `class`, а потому генерируют атрибуты, присоединяемые к классу — `FirstClass.setdata` и `FirstClass.display`. В действительности любое имя, присвоенное на верхнем уровне вложенного блока класса, становится атрибутом этого класса.

Функции внутри класса, как правило, называются *методами*. Они создаются посредством нормальных операторов `def` и поддерживают все, что вам уже известно о функциях (т.е. могут иметь стандартные значения аргументов, возвращать значения, выдавать элементы по запросу и т.п.). Но первый аргумент в функции метода при ее вызове автоматически получает подразумеваемый объект экземпляра — объект, на котором произведен вызов. Нам необходимо создать пару экземпляров, чтобы посмотреть, как все работает:

```
>>> x = FirstClass()     # Создать два экземпляра
>>> y = FirstClass()     # Каждый представляет собой новое пространство имен
```

Обращаясь к классу таким способом (обратите внимание на круглые скобки), мы генерируем объекты экземпляров, представляющие собой просто пространства имен, которые имеют доступ к атрибутам своих классов. Собственно говоря, в этой точке мы имеем три объекта: два экземпляра и класс. В действительности мы располагаем тремя связанными пространствами имен, как иллюстрируется на рис. 27.1. В терминах ООП мы говорим, что экземпляр `x` "является" `FirstClass`, равно как и `y` – они оба наследуют имена, присоединенные к классу.

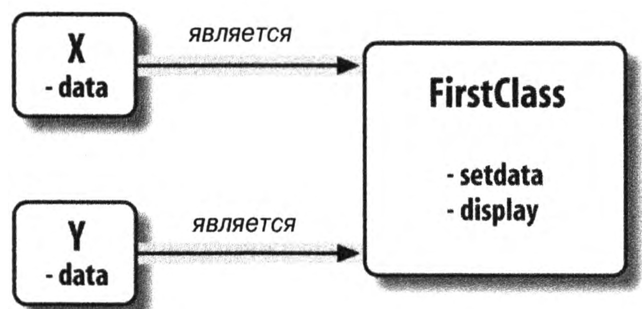


***Рис. 27.1.** Классы и экземпляры связывают объекты пространств имен в дерево классов, в котором ищутся атрибуты при поиске в иерархии наследования. Здесь атрибут `data` находится в экземплярах, но `setdata` и `display` присутствуют в классах, расположенных выше них*

Два экземпляра начинают свое существование как пустые, но имеют ссылки на класс, из которого они были сгенерированы. Если мы уточним экземпляр с помощью имени атрибута, который находится в объекте класса, тогда Python извлечет имя из класса посредством поиска в иерархии наследования (при условии, что он также не присутствует в экземпляре):

```
>>> x.setdata("King Arthur")    # Вызвать методы: self - это x
>>> y.setdata(3.14159)          # Выполняется FirstClass.setdata(y, 3.14159)
```

Ни `x`, ни `y` не имеет собственного атрибута `setdata`, поэтому чтобы найти его, Python следует по ссылке из экземпляра в класс. Вот и все, что нужно для наследования в Python: оно происходит во время уточнения атрибутов и предусматривает лишь поиск имен в связанных объектах – в данном случае за счет следования по ссылкам "является" на рис. 27.1.

В функции `setdata` класса `FirstClass` передаваемое значение присваивается `self.data`. Внутри метода `self` (имя, по соглашению назначаемое крайнему слева аргументу) автоматически ссылается на обрабатываемый экземпляр (`x` или `y`), так что присваивания сохраняют значения в пространствах имен экземпляров, а не класса; подобным образом создавались имена `data` на рис. 27.1.

Поскольку классы способны генерировать множество экземпляров, методы должны с помощью аргумента `self` получать обрабатываемый экземпляр. Вызвав метод `display` класса для вывода `self.data`, мы заметим, что он отличается для каждого экземпляра; с другой стороны, само имя `display` одинаковое в `x` и `y`, т.к. оно поступает (наследуется) от класса:

```
>>> x.display()    # self.data отличается в каждом экземпляре
King Arthur
>>> y.display()    # Выполняется FirstClass.display(y)
3.14159
```

Обратите внимание, что в каждом экземпляре мы сохраняем в члене data объекты разных типов – строку и число с плавающей точкой. Как и со всем остальным в Python, для атрибутов экземпляров (иногда называемых *членами*) не предусмотрено каких-либо объявлений; подобно простым переменным они появляются при первом присваивании значений. На самом деле, если бы мы вызвали display на одном из наших экземпляров *перед* вызовом setdata, то инициировали бы ошибку неопределенного имени – атрибут по имени data не существует в памяти до тех пор, пока не будет присвоен внутри метода setdata.

В качестве еще одного способа оценить, насколько динамична эта модель, имейте в виду, что мы можем изменять атрибуты в самом классе, присваивая self в методах, или *за пределами* класса путем присваивания явному объекту экземпляра:

```
>>> x.data = "New value"    # Можно получать/устанавливать атрибуты
>>> x.display()             # И за пределами класса тоже
New value
```

Хотя и менее часто, мы могли бы генерировать совершенно *новый* атрибут в пространстве имен экземпляра, присваивая значение его имени за пределами функций методов класса:

```
>>> x.anothername = "spam"  # Здесь можно также устанавливать новые атрибуты!
```

Такой оператор присоединит к объекту экземпляра x новый атрибут по имени anothername, который может применяться или нет любым методом класса. Классы обычно создают все атрибуты экземпляра путем присваивания аргумента self, но они не обязаны поступать так – программы могут извлекать, изменять или создавать атрибуты для любых объектов, ссылками на которые они располагают.

Обычно не имеет смысла добавлять данные, которыми класс не в состоянии пользоваться, и это можно предотвратить с помощью добавочного кода "защиты", основанного на перегрузке операции доступа к атрибутам, как будет обсуждаться позже в книге (в главах 30 и 39). Тем не менее, свободный доступ к атрибутам приводит к снижению сложности синтаксиса и есть ситуации, когда он даже полезен – например, при реализации разновидности записей данных, которые будут демонстрироваться позже в главе.

Классы настраиваются через наследование

Давайте перейдем ко второму крупному отличию классов. Помимо того, что классы служат фабриками для генерирования множества объектов экземпляров, они также предоставляют возможность вносить изменения за счет ввода новых компонентов (называемых *подклассами*) вместо изменения существующих компонентов на месте.

Как мы уже видели, объекты экземпляров, сгенерированные из класса, наследуют атрибуты этого класса. Python также позволяет классам быть унаследованными от других классов, открывая возможность создания *иерархий* классов, которые специализируют поведение. Переопределяя атрибуты в подклассах, которые находятся ниже в иерархии, мы переопределяем более общие определения таких атрибутов выше в дереве. По сути, чем ниже мы углубляемся в иерархию, тем более специфическим становится программное обеспечение. Здесь отсутствуют какие-либо параллели с модулями, чьи атрибуты находятся в единственном плоском пространстве имен, которое не поддается настройке.

Экземпляры в Python наследуют от классов, а классы – от суперклассов. Ниже описаны ключевые идеи, лежащие в основе механизма наследования атрибутов.

- Суперклассы перечисляются внутри круглых скобок в заголовке `class`. Чтобы заставить класс наследовать атрибуты от другого класса, просто укажите другой класс внутри круглых скобок в строке заголовка нового оператора `class`. Класс, выполняющий наследование, обычно называется подклассом, а класс, от которого производится наследование, является его суперклассом.
- Классы наследуют атрибуты от своих суперклассов. Точно так же, как экземпляры наследуют имена атрибутов, определенные в их классах, классы наследуют все имена атрибутов, которые определены в их суперклассах; при доступе к атрибутам Python находит их автоматически, если они не существуют в подклассах.
- Экземпляры наследуют атрибуты от всех доступных классов. Каждый экземпляр получает имена от класса, из которого он сгенерирован, а также от всех суперклассов этого класса. При поиске имени Python проверяет экземпляр, затем его класс и, наконец, все суперклассы.
- Каждая ссылка `объект.атрибут` инициирует новый независимый поиск. Python выполняет независимый поиск в дереве классов для каждого выражения с извлечением атрибута. Сюда входят ссылки на экземпляры и классы, сделанные за пределами операторов `class` (например, `X.атрибут`), а также ссылки на атрибуты экземпляра аргумента `self` в функциях методов класса. Каждое выражение `self.атрибут` в методе вызывает новый поиск для `атрибута` в `self` и выше.
- Изменения в логику вносятся за счет создания подклассов, а не модификации суперклассов. Переопределяя имена суперклассов в подклассах ниже в иерархии (дерева классов), подклассы замещают и тем самым настраивают унаследованное поведение.

Совокупный эффект — и основная цель всего поиска подобного рода — заключается в том, что классы лучше поддерживают разложение на составляющие и настройку кода, чем другие языковые инструменты, рассмотренные до сих пор. С одной стороны, они позволяют нам свести к минимуму избыточность кода (и в итоге сократить расходы на сопровождение), вынося операции в единственную разделяемую реализацию. С другой стороны, они дают нам возможность программировать путем настройки того, что уже существует, а не его изменения на месте или написания кода с нуля.

Строго говоря, *наследование* Python оказывается чуть более развитым, чем здесь описано, когда мы задействуем дескрипторы нового стиля и метаклассы (сложные темы, исследуемые позже), но мы можем благополучно ограничиваться экземплярами и их классами, как в этом месте книги, так и в большинстве прикладного кода Python. Формально наследование определяется в главе 40.

Второй пример

Чтобы подкрепить иллюстрацией роль наследования, текущий пример будет построен на основе предыдущего. Первым делом мы определим новый класс `SecondClass`, который наследует все имена `FirstClass` и предоставляет одно собственное имя:

```
>>> class SecondClass(FirstClass):                # Наследует setdata
        def display(self):                        # Изменяет display
            print('Current value = "%s"' % self.data)
```

Класс SecondClass определяет метод display для вывода в другом формате. За счет определения атрибута с таким же именем, как у атрибута в FirstClass, класс SecondClass фактически замещает атрибут display в своем суперклассе.

Вспомните, что поиск в иерархии наследования направлен вверх от экземпляров к подклассам и далее к суперклассам, останавливаясь на первом найденном вхождении имени атрибута. В данном случае, поскольку имя display в SecondClass будет обнаружено перед таким же именем в FirstClass, мы говорим, что SecondClass переопределяет display из FirstClass. Действие по замещению атрибутов путем их переопределения ниже в дереве иногда называют *перегрузкой*.

Конечный результат здесь в том, что SecondClass специализирует FirstClass за счет изменения поведения метода display. С другой стороны, класс SecondClass (и любые созданные из него экземпляры) по-прежнему наследует метод setdata от FirstClass буквально. Давайте в целях демонстрации создадим экземпляр:

```
>>> z = SecondClass()
>>> z.setdata(42)          # Находит setdata в FirstClass
>>> z.display()            # Находит переопределенный метод в SecondClass
Current value = "42"
```

Как и ранее, мы создаем объект экземпляра SecondClass посредством обращения к нему. Вызов setdata приводит к выполнению версии из FirstClass, но атрибут display на этот раз поступает из SecondClass и выводит специальное сообщение. На рис. 27.2 показаны задействованные пространства имен.

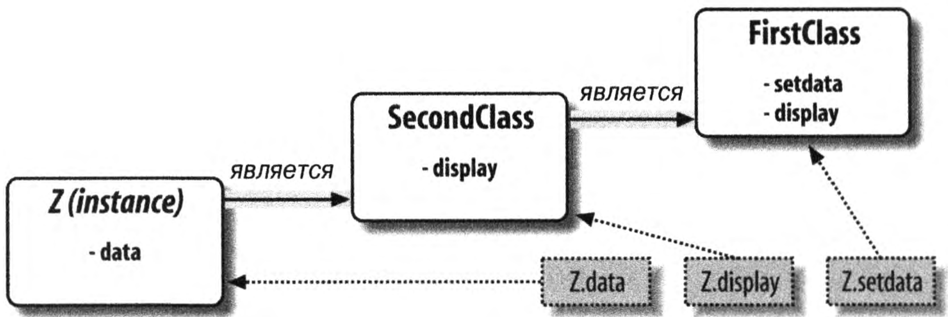


Рис. 27.2. *Специализация: переопределение унаследованных имен за счет их повторного определения в расширениях ниже в дереве классов. Здесь SecondClass переопределяет и тем самым настраивает метод display для своих экземпляров*

Касательно ООП важно отметить один важный момент: специализация, введенная в SecondClass, является полностью *внешней* по отношению к FirstClass. Таким образом, она не затрагивает существующие или будущие объекты FirstClass, подобные x из предыдущего примера:

```
>>> x.display() # x - по-прежнему экземпляр FirstClass (выводит старое сообщение)
New value
```

Вместо *изменения* класса FirstClass мы *настроили* его. Естественно, это искусственный пример, но в качестве правила запомните, что поскольку наследование дает возможность вносить изменения такого рода во внешние компоненты (т.е. в подклассы), классы часто поддерживают расширение и многократное использование лучше, чем функции или модули.

Классы являются атрибутами в модулях

Прежде чем двигаться дальше, следует отметить, что с именем класса не связано ничего магического. Оно представляет собой всего лишь переменную, которой при выполнении оператора class присваивается объект, и на объект можно ссылаться с помощью любого нормального выражения. Например, если бы вместо набора в интерактивной подсказке класс FirstClass был помещен в файл модуля, тогда мы могли бы импортировать его и применять его имя обычным образом в строке заголовка class:

```
from modulename import FirstClass   # Копировать имя в текущую область видимости
class SecondClass(FirstClass):      # Использовать имя класса напрямую
    def display(self): ...
```

Или вот эквивалент:

```
import modulename                            # Доступ к целому модулю
class SecondClass(modulename.FirstClass):    # Уточнение для ссылки
    def display(self): ...
```

Как и все остальное, имена классов всегда существуют внутри модуля, так что они должны следовать всем правилам, которые мы обсуждали в части V. Например, в единственном файле модуля может находиться несколько классов — подобно другим операторам в модуле операторы class выполняются во время импортирования для определения имен, которые становятся индивидуальными атрибутами модуля. В более общем случае каждый модуль может произвольно смешивать любое количество переменных, функций и классов, причем все имена в модуле ведут себя одинаково. В целях демонстрации ниже приведено содержимое food.py:

```
# food.py
var = 1              # food.var
def func(): ...      # food.func
class spam: ...      # food.spam
class ham: ...       # food.ham
class eggs: ...      # food.eggs
```

Сказанное остается справедливым, даже если модуль и класс имеют совпадающие имена. Например, при наличии файла person.py со следующим содержимым:

```
class person: ...
```

для извлечения класса необходимо обычным образом указать модуль:

```
import person                 # Импортировать модуль
x = person.person()           # Класс внутри модуля
```

Несмотря на то что такой путь может выглядеть избыточным, он обязателен: person.person ссылается на класс person внутри модуля person. Указание только person приводит к получению модуля, но не класса, если только не используется оператор from:

```
from person import person     # Получить класс из модуля
x = person()                  # Использовать имя класса
```

Как в случае любой другой переменной, мы не можем увидеть класс в файле без предварительного импортирования и его извлечения из включающего файла. Если это кажется непонятным, тогда не применяйте одинаковые имена для модуля и класса внутри него. На самом деле по соглашению, принятому в Python, имена классов должны начинаться с буквы *верхнего регистра*, чтобы сделать их более различимыми:

```
import person                    # Нижний регистр для имен модулей
x = person.Person()              # Верхний регистр для имен классов
```

Кроме того, имейте в виду, что хотя и классы, и модули являются пространствами имен для присоединения атрибутов, они соответствуют очень разным структурам исходного кода: модуль отражает целый *файл*, а класс является *оператором* внутри файла. Мы обсудим такие отличия позже в данной части книги.

Классы могут перехватывать операции Python

Давайте перейдем к рассмотрению третьего и последнего отличия между классами и модулями: перегрузке операций. Используя простые термины, *перегрузка операций* позволяет объектам, сознанным из классов, перехватывать и реагировать на операции, которые работают со встроенными типами: сложение, нарезание, вывод, уточнение и т.д. По большей части это просто механизм автоматической диспетчеризации — выражения и другие встроенные операции передают управление реализациям в классах. Здесь тоже нет ничего схожего с модулями: модули могут реализовывать вызовы функций, но не поведение выражений.

Несмотря на возможность реализации всего поведения класса в виде функций методов, перегрузка операций позволяет объектам более тесно интегрироваться с объектной моделью Python. Кроме того, поскольку перегрузка операций заставляет наши объекты действовать подобно встроенным объектам, это способствует созданию более согласованных и легких в изучении объектных интерфейсов, а также делает возможной обработку объектов, основанных на классах, с помощью кода, который написан в расчете на интерфейс встроенного типа. Ниже приведено краткое изложение главных идей, лежащих в основе перегрузки операций.

- Методы, имена которых содержат удвоенные символы подчеркивания (`__X__`), являются специальными привязками. В классах Python мы реализуем перегрузку операций за счет предоставления особым образом именованных методов для перехвата операций. В языке Python определено фиксированное и неизменяемое отображение каждой операции на метод со специальным именем.
- Такие методы вызываются автоматически, когда экземпляры встречаются во встроенных операциях. Скажем, если объект экземпляра наследует метод `__add__`, то этот метод вызывается всякий раз, когда объект появляется в выражении с операцией +. Возвращаемое значение метода становится результатом соответствующего выражения.
- Классы могут переопределять большинство встроенных операций с типами. Существуют десятки специальных имен методов для перегрузки операций, которые можно перехватывать и реализовывать почти каждую операцию, действующую на встроенных типах. Сюда входят не только операции выражений, но также базовые операции наподобие вывода и создания объектов.
- Для методов перегрузки операций не предусмотрены стандартные реализации и ни один из них не является обязательным. Если класс не определяет или не наследует какой-то метод перегрузки операции, то это просто означает, что соответствующая операция не поддерживается для экземпляров класса. Например, если метод `__add__` отсутствует, тогда выражения + будут приводить к исключениям.

- Классы нового стиля имеют ряд стандартных реализаций, но не для распространенных операций. В Python 3.X и в так называемых классах "нового стиля" из Python 2.X, которые мы определим позже, корневой класс по имени `object` предоставляет стандартные реализации для нескольких методов `__X__`, но их немного, и они не относятся к числу наиболее часто применяемых операций.
- Операции позволяют интегрировать классы в объектную модель Python. За счет перегрузки операций для типов определяемые пользователем объекты, которые мы реализуем посредством классов, могут действовать в точности как встроенные типы и потому обеспечивать согласованность, а также совместимость с ожидаемыми интерфейсами.

Перегрузка операций является необязательной возможностью; она используется разработчиками инструментов для других программистов на Python, а не разработчиками прикладных приложений. Откровенно говоря, вероятно вы не должны применять перегрузку операций лишь потому, что это выглядит умным или "крутым". Если класс не нуждается в имитации интерфейсов встроенных типов, то обычно необходимо придерживаться более просто именованных методов. Скажем, зачем приложению, работающему с базой данных сотрудников, поддерживать выражения вроде `*` и `+`? Именованные методы, подобные `giveRaise` и `promote`, как правило, будут иметь гораздо больший смысл.

Таким образом, мы не будем вдаваться в детали каждого метода перегрузки операции, доступного в Python. Однако имеется один метод перегрузки операции, который вы наверняка встретите почти в любом реалистичном классе Python: метод `__init__`, известный как метод *конструктора* и используемый для инициализации состояния объектов. Методу `__init__` должно уделяться особое внимание, поскольку наряду с аргументом `self` он оказывается ключевым условием для чтения и понимания большинства объектно-ориентированного кода на Python.

Третий пример

Рассмотрим еще один пример. На этот раз мы определим подкласс класса `SecondClass` из предыдущего раздела и реализуем три особым образом именованных атрибута, которые будут автоматически вызываться Python:

- `__init__` выполняется, когда создается новый объект экземпляра: `self` является новым объектом `ThirdClass`[1];
- `__add__` выполняется, когда экземпляр `ThirdClass` присутствует в выражении `+`;
- `__str__` выполняется, когда объект выводится (формально при его преобразовании в отображаемую строку встроенной функцией `str` или ее внутренним эквивалентом Python).

В новом подклассе также определен нормально именованный метод `mul`, который изменяет объект экземпляра на месте. Вот код нового подкласса:

```
>>> class ThirdClass(SecondClass):           # Унаследован от SecondClass
        def __init__(self, value):           # Вызывается для ThirdClass(value)
            self.data = value
        def __add__(self, other):            # Вызывается для self + other
            return ThirdClass(self.data + other)
```

[1] Не путайте его с файлами `__init__.py` в пакетах модулей! Метод здесь представляет собой функцию конструктора класса, применяемую для инициализации вновь созданного экземпляра, а не пакета модуля. Дополнительные сведения ищите в главе 24.

```
        def __str__(self):      # Вызывается для print(self), str()
            return '[ThirdClass: %s]' % self.data
        def mul(self, other):   Изменение на месте: именованный метод
            self.data *= other
>>> a = ThirdClass('abc')       # Вызывается __init__
>>> a.display()                 # Вызывается унаследованный метод
Current value = "abc"
>>> print(a)                    # __str__: возвращает отображаемую строку
[ThirdClass: abc]

>>> b = a + 'xyz'               # __add__: создает новый экземпляр
>>> b.display()                 # b имеет все методы класса ThirdClass
Current value = "abcxyz"

>>> print(b)                    # __str__: возвращает отображаемую строку
[ThirdClass: abcxyz]

>>> a.mul(3)                    # mul: изменяет экземпляр на месте
>>> print(a)
[ThirdClass: abcabcabc]
```

Класс ThirdClass "является" SecondClass, поэтому его экземпляры наследуют настроенный метод display от класса SecondClass из предыдущего раздела. Тем не менее, на этот раз при создании экземпляра ThirdClass передается аргумент ('abc'). Аргумент передается аргументу value конструктора __init__ и присваивается здесь атрибуту self.data. Совокупный эффект в том, что ThirdClass организован так, чтобы устанавливать атрибут data автоматически во время конструирования, не требуя последующего вызова setdata.

Кроме того, объекты ThirdClass теперь могут появляться в выражениях + и вызовах print. В случае выражения + интерпретатор Python передает объект экземпляра слева аргументу self и значение справа аргументу other в методе __add__ (рис. 27.3); любое возвращаемое значение __add__ становится результатом выражения + (вскоре мы более подробно обсудим его результат).

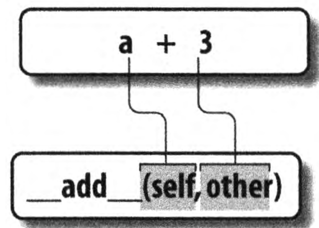


Рис. 27.3. При перегрузке операции выражений и другие встроенные операции, выполняемые над классом, отображаются на методы с особыми именами в классе. Такие специальные методы необязательны и могут быть унаследованы обычным образом. В данном случае выражение + запускает метод __add__

Для вызова print интерпретатор Python передает выводимый объект аргументу self в методе __str__; любая возвращаемая этим методом строка становится отображаемой строкой для объекта. Благодаря методу __str__ (или его более подходящему двойнику __repr__, который мы будем использовать в следующей главе) мы можем применять для вывода объектов данного класса нормальный вызов print вместо обращения к специальному методу display.

Особым образом именованные методы, такие как __init__, __add__ и __str__, наследуются подклассами и экземплярами в точности подобно любым другим именам,

присваиваемым в операторе class. Если они не реализованы в классе, тогда Python ищет такие имена во всех суперклассах класса, как обычно. Имена методов для перегрузки операций также не являются встроенными или зарезервированными словами; они представляют собой всего лишь атрибуты, которые Python ищет, когда объекты появляются в разнообразных контекстах. Обычно Python вызывает их автоматически, но иногда они могут вызываться также и в вашем коде. Например, как мы увидим в следующей главе, метод `__init__` часто вызывается вручную для запуска шагов инициализации в суперклассе.

Возвращать результаты или нет

Некоторые методы для перегрузки операций, скажем, `__str__`, требуют результатов, но другие обладают большей гибкостью. Например, обратите внимание на то, как метод `__add__` создает и возвращает *новый* объект экземпляра класса ThirdClass, вызывая ThirdClass с результирующим значением, что в свою очередь запускает `__init__` для инициализации результатом. Это общее соглашение, которое объясняет, почему переменная b в листинге имеет метод display; она тоже является объектом ThirdClass, т.к. именно его возвращает операция + для объектом данного класса. По существу тип становится более распространенным.

И напротив, метод mul *изменяет* текущий объект экземпляра на месте, заново присваивая атрибут self. Чтобы сделать последнее, мы могли бы перегрузить операцию выражения *, но тогда результат слишком бы отличался от поведения * для встроенных типов, таких как числа и строки, где операция * всегда создает новые объекты. Общая практика требует, чтобы перегруженные операции работали таким же образом, как их встроенные реализации. Однако поскольку перегрузка операций в действительности представляет собой просто механизм диспетчеризации между выражениями и методами, в объектах собственных классов вы можете интерпретировать операции любым желаемым способом.

Для чего используется перегрузка операций?

Как проектировщик класса, вы сами решаете, применять перегрузку операций или нет. Выбор зависит просто от того, насколько вы хотите, чтобы ваш объект был похож по виду и поведению на встроенные типы. Как упоминалось ранее, если опустить метод перегрузки операции и не наследовать его от суперкласса, тогда соответствующая операция для экземпляров поддерживаться не будет; при попытке ее использовать возникнет исключение (или в некоторых случаях вроде вывода будет применяться стандартная реализация).

По правде говоря, многие методы для перегрузки операций, как правило, применяются при реализации объектов, имеющих математическую природу; скажем, класс вектора или матрицы может перегружать операцию сложения, но класс сотрудника — вряд ли. Для более простых классов вы можете вообще не использовать перегрузку и при реализации поведения объектов взамен полагаться на явные вызовы методов.

С другой стороны, вы можете принять решение применять перегрузку операция, если определяемые пользователем объекты необходимо передавать функции, которая написана так, что ожидает операций, доступных для встроенного типа вроде списка или словаря. Реализация в классе того же самого набора операций будет гарантировать, что ваши объекты поддерживают такой же ожидаемый объектный интерфейс, а потому совместимы с функцией. Хотя мы не станем раскрывать в книге каждый метод для перегрузки операций, в главе 30 будет дан обзор дополнительных распространенных методик перегрузки операций.

Одним из методов перегрузки, который мы будем часто здесь использовать, является метод конструктора `__init__`, применяемый для инициализации вновь созданных объектов экземпляров и присутствующий почти в каждом реалистичном классе. Из-за того, что конструктор позволяет классам немедленно заполнять атрибуты в своих новых экземплярах, он удобен практически во всех видах классов, которые вам доведется реализовывать. На самом деле, хотя атрибуты экземпляра в Python не объявляются, обычно легко выяснить, какие атрибуты будет иметь экземпляр, проинспектировав метод `__init__` его класса.

Разумеется, нет ничего плохого в том, чтобы проводить эксперименты с интересными языковыми инструментами, но они не всегда переносятся в производственный код. С течением времени и накоплением опыта вы начнете считать такие программные структуры и указания естественными и чуть ли не автоматическими.

Простейший в мире класс Python

В этой главе мы начали подробное исследование синтаксиса оператора `class`, но я хотел бы напомнить еще раз, что базовая модель наследования, которую производят классы, очень проста — в действительности она включает в себя всего лишь поиск атрибутов в деревьях связанных объектов. Фактически мы можем создать класс, не содержащий вообще ничего. Следующий оператор создает класс без присоединенных атрибутов, т.е. объект пустого пространства имен:

```
>>> class rec: pass          # Объект пустого пространства имен
```

Оператор заполнителя `pass` (обсуждаемый в главе 13) здесь нужен потому, что в классе нет ни одного метода. После создания класса путем запуска показанного выше оператора в интерактивной подсказке мы можем заняться присоединением атрибутов к классу, присваивая его именам значения полностью за пределами исходного оператора `class`:

```
>>> rec.name = 'Bob'         # Просто объект с атрибутами
>>> rec.age = 40
```

Создав с помощью присваивания атрибуты, мы можем извлекать их с использованием обычного синтаксиса. В случае применения подобным образом класс напоминает "структуру" в C или "запись" в Pascal. По существу это объект с присоединенными к нему именами полей (как мы увидим далее, похожий трюк с ключами словаря требует набора дополнительных символов):

```
>>> print(rec.name)          # Подобен структуре C или записи
Bob
```

Обратите внимание, что прием работает, даже когда еще нет *ни одного экземпляра* класса; классы сами по себе являются объектами и без экземпляров. В действительности они представляют собой всего лишь автономные пространства имен; до тех пор, пока у нас есть ссылка на класс, мы в любой момент можем устанавливать либо изменять его атрибуты. Тем не менее, взгляните, что происходит, когда мы создаем два экземпляра:

```
>>> x = rec()                # Экземпляры наследуют имена класса
>>> y = rec()
```

Экземпляры начинают свое существование как объекты совершенно пустых пространств имен. Однако поскольку экземпляры запоминают класс, из которого были

созданы, они будут извлекать атрибуты, присоединенные нами к классу, через наследование:

```
>>> x.name, y.name    # name хранится только в классе
('Bob', 'Bob')
```

На самом деле эти экземпляры сами не имеют атрибутов; они просто извлекают атрибут name из объекта класса, где он хранится. Если же мы присваиваем атрибуту экземпляра, тогда создается (или изменяется) атрибут в одном объекте, но не в другом — критически важно то, что *ссылки* на атрибуты инициируют поиск в иерархии наследования, а *присваивания* атрибутов влияют только на объекты, в которых присваивания выполнялись. Здесь это означает, что x получает собственный атрибут name, но y по-прежнему наследует атрибут name, присоединенный к классу выше в дереве:

```
>>> x.name = 'Sue'    # Но присваивание изменяет только x
>>> rec.name, x.name, y.name
('Bob', 'Sue', 'Bob')
```

На самом деле, как будет более детально исследовано в главе 29, атрибуты объекта пространства имен обычно реализуются как словари, а деревья наследования классов представляют собой (говоря в общем) всего лишь словари, содержащие связи с другими словарями. Если вы знаете, куда смотреть, то сможете увидеть это явно.

Например, атрибут __dict__ является словарем пространств имен для большинства объектов, основанных на классах. Ряд классов могут дополнительно (или взамен) определять атрибуты в __slots__ — расширенное и редко используемое средство, которое упоминается в главе 28, но будет более детально рассматриваться в главах 31 и 32. Обычно __dict__ буквально представляет собой пространство имен атрибутов экземпляра.

В целях иллюстрации ниже приведено взаимодействие в Python 3.7; порядок следования имен и набор внутренних имен __X__ могут варьироваться от выпуска к выпуску, к тому же мы отфильтровали встроенные имена с помощью генераторного выражения, как поступали ранее, но все присвоенные нами имена присутствуют:

```
>>> list(rec.__dict__.keys())
['__module__', '__dict__', '__weakref__', '__doc__', 'name', 'age']

>>> list(name for name in rec.__dict__ if not name.startswith('__'))
['age', 'name']
>>> list(x.__dict__.keys())
['name']
>>> list(y.__dict__.keys())    # list() не требуется в Python 2.X
[]
```

Здесь словарь пространств имен класса содержит присвоенные ранее атрибуты name и age, экземпляр x имеет собственный атрибут name, а экземпляр y все еще пуст. Из-за такой модели атрибут часто может извлекаться *либо* посредством индексирования словаря, *либо* с помощью записи атрибута, но только если он присутствует в обрабатываемом объекте. Запись атрибута инициирует поиск в иерархии наследования, но индексирование ищет *только* в одиночном объекте (как будет показано позже, оба подхода исполняют допустимые роли):

```
>>> x.name, x.__dict__['name']  # Представленные здесь атрибуты являются
                                #   ключами словаря
('Sue', 'Sue')
>>> x.age                       # Но извлечение атрибута проверяет также классы
40
```

```
>>> x.__dict__['age']            # Индексирование словаря не производит поиск
                                 #   в иерархии наследования
KeyError: 'age'
Ошибка ключа: 'age'
```

Для упрощения поиска в иерархии наследования при извлечении атрибутов каждый экземпляр имеет связь со своим классом, которую создает Python — она называется `__class__` и ее можно просмотреть:

```
>>> x.__class__                  # Связь экземпляра с классом
<class '__main__.rec'>
```

Классы также располагают атрибутом `__bases__`, который является кортежем ссылок на их объекты суперклассов — в данном примере только подразумеваемый корневой класс `object` в Python 3.X, исследуемый позже (в Python 2.X взамен получается пустой кортеж):

```
>>> rec.__bases__                # Связь с суперклассами, () в Python 2.X
(<class 'object'>,)
```

Эти два атрибута показывают, каким образом деревья классов буквально представлены в памяти. Внутренние детали подобного рода знать необязательно (деревья классов вытекают из выполняемого кода), но они часто помогают прояснить модель.

Главное, на что стоит обратить внимание — модель классов Python чрезвычайно динамична. Классы и экземпляры представляют собой всего лишь объекты пространств имен с атрибутами, создаваемыми на лету через присваивания. Такие присваивания, как правило, происходят внутри записываемых вами операторов `class`, но могут встречаться везде, где имеется ссылка на один из объектов в дереве.

Даже *методы*, которые обычно создаются с помощью операторов `def`, вложенных в `class`, могут быть созданы совершенно независимо от любого объекта класса. Например, следующий код определяет простую функцию, принимающую один аргумент, за пределами любого класса:

```
>>> def uppername(obj):
        return obj.name.upper()   # По-прежнему необходим аргумент self (obj)
```

Здесь пока еще ничего не связано с классом; `uppername` является простой функцией и может вызываться в данной точке при условии передачи ей объекта `obj` с атрибутом `name`, значение которого имеет метод `upper`. Экземпляры нашего класса соответствуют ожидаемому интерфейсу и запускают преобразование строк в верхний регистр:

```
>>> uppername(x)                 # Вызов как простой функции
'SUE'
```

Тем не менее, если мы присвоим эту простую функцию атрибуту нашего класса, она становится *методом*, допускающим вызов через любой экземпляр, а также через имя самого класса при условии передачи экземпляра вручную — методика, которую мы задействуем в следующей главе[2]:

---

[2] На самом деле это одна из причин, по которым аргумент `self` *обязан* всегда явно присутствовать в методах Python — поскольку методы могут создаваться как простые функции, независимые от класса, они должны делать явным аргумент подразумеваемого экземпляра. Их можно вызывать либо как функции, либо как методы, и Python не может ни угадать, ни предположить о том, что простая функция в конечном итоге станет методом класса. Однако главная причина явного указания аргумента `self` заключается в том, чтобы сделать смысл имен более очевидным. Имена, на которые производится ссылка через `self`, представляют собой простые переменные, отображаемые на области видимости, тогда как имена, на которые ссылаются через `self` с помощью записи атрибутов, совершенно ясно являются атрибутами экземпляра.

```
>>> rec.method = uppername       # Теперь это метод класса!
>>> x.method()                   # Запустить метод для обработки x
'SUE'
>>> y.method()                   # То же самое, но передать y для self
'BOB'
>>> rec.method(x)                # Можно вызывать через экземпляр или класс
'SUE'
```

Обычно классы заполняются посредством операторов `class`, а атрибуты экземпляров создаются с помощью присваиваний атрибутам `self` в функциях методов. Однако мы еще раз отметим, что поступать так необязательно; ООП в Python главным образом касается поиска атрибутов в связанных объектах пространств имен.

Снова о записях: классы или словари

Хотя простые классы в предыдущем разделе были предназначены для иллюстрации основ модели классов, те методики, которые в них задействованы, могут также применяться в реальной работе. Скажем, в главах 8 и 9 демонстрировалось использование словарей, кортежей и списков для хранения в программах свойств сущностей, в общем случае называемых *записями*. Оказывается, что классы способны быть более эффективными в такой роли – они упаковывают информацию подобно словарям, но могут также умещать в себе логику обработки в форме методов. Для справочных целей ниже приведен пример записей на основе кортежа и словаря, которые применялись ранее в книге (здесь используется один из многочисленных приемов реализации словарей):

```
>>> rec = ('Bob', 40.5, ['dev', 'mgr']) # Запись на основе кортежа
>>> print(rec[0])
Bob

>>> rec = {}
>>> rec['name'] = 'Bob'                 # Запись на основе словаря
>>> rec['age'] = 40.5                   # Или {...}, dict(n=v) и т.д.
>>> rec['jobs'] = ['dev', 'mgr']
>>>
>>> print(rec['name'])
Bob
```

Код эмулирует инструменты, похожие на записи в других языках. Тем не менее, как только что выяснилось, существует также множество способов делать то же самое с применением классов. Пожалуй, простейший из них предусматривает замену ключей атрибутами:

```
>>> class rec: pass
>>> rec.name = 'Bob'                    # Запись на основе класса
>>> rec.age = 40.5
>>> rec.jobs = ['dev', 'mgr']
>>>
>>> print(rec.name)
Bob
```

Показанный выше код существенно меньше, чем эквивалент в виде словаря. В нем с помощью оператора `class` создается объект пустого пространства имен. С течением времени полученный пустой класс заполняется путем присваивания атрибутов класса, как и ранее.

Прием работает, но для каждой отличающейся записи будет требоваться новый оператор class. Вероятно, более естественно взамен генерировать *экземпляры* пустого класса для представления каждой отличающейся записи:

```
>>> class rec: pass

>>> pers1 = rec()                            # Записи на основе экземпляров
>>> pers1.name = 'Bob'
>>> pers1.jobs = ['dev', 'mgr']
>>> pers1.age = 40.5
>>>
>>> pers2 = rec()
>>> pers2.name = 'Sue'
>>> pers2.jobs = ['dev', 'cto']
>>>
>>> pers1.name, pers2.name
('Bob', 'Sue')
```

Здесь мы создали две записи из одного и того же класса. Как и классы, экземпляры начинают свое существование пустыми. Затем мы заполняем записи, делая присваивания их атрибутам. Однако на этот раз существуют два отдельных объекта, а потому два разных атрибута name. Фактически экземпляры того же самого класса даже не обязаны иметь одинаковые наборы имен атрибутов; в приведенном примере один из них располагает уникальным именем age. Экземпляры в действительности являются отличающимися пространствами имен, так что каждый имеет отдельный словарь атрибутов. Хотя обычно экземпляры согласованно заполняются методами класса, они намного гибче, чем можно было бы ожидать.

Наконец, мы можем вместо этого создать более развитый класс с целью реализации записи *и* ее обработки — то, что словари, ориентированные на данные, не поддерживают напрямую:

```
>>> class Person:
        def __init__(self, name, jobs, age=None): # Класс = данные + логика
            self.name = name
            self.jobs = jobs
            self.age = age
        def info(self):
            return (self.name, self.jobs)

>>> rec1 = Person('Bob', ['dev', 'mgr'], 40.5)    # Вызовы конструктора
>>> rec2 = Person('Sue', ['dev', 'cto'])
>>>
>>> rec1.jobs, rec2.info()                        # Атрибуты + методы
(['dev', 'mgr'], ('Sue', ['dev', 'cto']))
```

Такая схема также создает множество экземпляров, но теперь класс не пустой: мы добавили *логику* (методы) для инициализации экземпляров при их создании и сбора атрибутов в кортеж по запросу. Конструктор придает экземплярам некоторую согласованность, всегда устанавливая атрибуты name, job и age, хотя последний атрибут может не указываться. Вместе методы класса и атрибуты экземпляра образуют *пакет*, объединяющий *данные* и *логику*.

Мы могли бы дальше расширять этот код, добавляя логику для расчета заработных плат, разбора имен и т.п. В конце концов, мы можем поместить класс в более крупную иерархию, чтобы наследовать и настраивать существующий набор методов через автоматический поиск атрибутов классов, или даже сохранять экземпляры класса в файле с помощью модуля pickle, обеспечивая их постоянство. На самом деле мы так

и *поступим* – в следующей главе рассмотренная аналогия между классами и записями будет расширена за счет реализации более реалистичного рабочего примера, который продемонстрирует основы классов в действии.

Чтобы отдать должное другим инструментам в показанной выше форме два вызова конструктора больше напоминают словари, созданные все за раз, но все-таки классы характеризуются меньшим беспорядком и предоставляют дополнительные методы обработки. В действительности вызовы конструктора класса больше похожи на *именованные кортежи* из главы 9 – это имеет смысл с учетом того, что именованные кортежи *являются* классами с добавочной логикой для отображения атрибутов на смещения кортежей:

```
>>> rec = dict(name='Bob', age=40.5, jobs=['dev', 'mgr'])        # Словари
>>> rec = {'name': 'Bob', 'age': 40.5, 'jobs': ['dev', 'mgr']}
>>> rec = Rec('Bob', 40.5, ['dev', 'mgr'])                # Именованные кортежи
```

В заключение отметим, что хотя типы вроде словарей и кортежей обладают гибкостью, классы позволяют нам добавлять к объектам поведение способами, которые не поддерживаются встроенными типами и простыми функциями напрямую. Несмотря на то что мы можем хранить функции в словарях, их использование для обработки подразумеваемых экземпляров оказывается далеко не таким естественным и структурированным, как это сделано в классах. Сказанное прояснится в следующей главе.

Резюме

В главе были представлены основы написания классов на Python. Вы изучили синтаксис оператора `class` и узнали, как его применять для построения дерева наследования классов. Вы также выяснили, как Python автоматически заполняет первый аргумент в функциях методов, как атрибуты присоединяются к объектам в дереве классов с помощью простого присваивания и как особым образом именованные методы для перегрузки операций перехватывают и реализовывают встроенные операции, работающие с нашими экземплярами (например, выражения и вывод).

Теперь, когда вы узнали все об особенностях создания классов в Python, в следующей главе мы займемся более крупным и реалистичным примером, в котором увязывается вместе большинство того, что было изучено в ООП до сих пор, и представим ряд новых тем. Затем мы продолжим исследование создания классов, сделав второй проход по модели, чтобы восполнить детали, которые ради простоты здесь были опущены. Но прежде чем двигаться дальше, закрепите пройденный материал главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Как классы связаны с модулями?
2. Каким образом создаются экземпляры и классы?
3. Где и как создаются атрибуты класса?
4. Где и как создаются атрибуты экземпляра?
5. Что `self` означает в классе Python?
6. Каким образом реализовывать перегрузку операций в классе Python?

7. Когда может понадобиться поддержка перегрузки операций в классах?
8. Какой метод перегрузки операции используется наиболее часто?
9. Какие две концепции обязательно знать для понимания объектно-ориентированного кода на Python?

Проверьте свои знания: ответы

1. Классы всегда вкладываются внутрь модуля; они являются атрибутами объекта модуля. Классы и модули являются пространствами имен, но классы соответствуют операторам (не целым файлам) и поддерживают такие понятия ООП, как множество экземпляров, наследование и перегрузку операций (все перечисленное модули не поддерживают). До известной степени модуль подобен классу с единственным экземпляром без наследования, который соответствует полному файлу кода.

2. Классы создаются путем выполнения операторов `class`; экземпляры создаются за счет обращения к классу, как если бы он был функцией.

3. Атрибуты класса создаются путем выполнения присваивания атрибутам объекта класса. Они обычно генерируются присваиваниями верхнего уровня, вложенными внутрь оператора `class` — каждое имя, присвоенное в блоке оператора `class`, становится атрибутом объекта класса (формально локальная область видимости оператора `class` превращается в пространство имен для атрибутов объекта класса, что во многом похоже на модуль). Тем не менее, атрибуты класса можно также создавать путем их присваивания везде, где имеется ссылка на объект класса — даже за пределами оператора `class`.

4. Атрибуты экземпляра создаются посредством присваивания значений атрибутам объекта экземпляра. Они обычно создаются в функциях методов класса, реализованных внутри оператора `class`, с помощью присваивания значений атрибутам аргумента `self` (который всегда является подразумеваемым экземпляром). Однако их тоже можно создавать присваиванием везде, где присутствует ссылка на экземпляр, даже за пределами оператора `class`. Обычно все атрибуты экземпляра инициализируются в методе конструктора `__init__`; таким образом, более поздние вызовы методов могут предполагать, что атрибуты уже существуют.

5. `self` — это имя, обычно назначаемое первому (крайнему слева) аргументу в функции метода класса; Python автоматически заполняет его объектом экземпляра, который представляет собой подразумеваемый объект вызова метода. Данный аргумент не обязан называться `self` (хотя соглашение очень строгое); важна его позиция. (Бывшие программисты на C++ или Java могут предпочесть назначать ему имя `this`, поскольку в языках C++ и Java такое имя отражает ту же самую идею; тем не менее, в Python этот аргумент должен всегда быть явным.)

6. Перегрузка операций реализуется в классе Python посредством особым образом именованных методов; имена начинаются и заканчиваются двумя символами подчеркивания, чтобы сделать их уникальными. Имена не являются встроенными или зарезервированными; Python всего лишь автоматически выполняет их, когда экземпляр встречается в соответствующей операции. Сам Python определяет отображения операций на специальные имена методов.

7. Перегрузка операций полезна для реализации объектов, которые имеют сходство со встроенными типами (например, последовательностей или числовых объектов, таких как матрицы), и для имитации интерфейса встроенного типа, ожидаемого порцией кода. Имитация интерфейсов встроенных типов дает возможность передавать экземпляры классов, которые также содержат информацию состояния (т.е. атрибуты, запоминающие данные между вызовами операции). Однако вы не должны применять перегрузку операций, когда будет достаточно простого именованного метода.

8. Наиболее часто используется метод конструктора `__init__`; почти каждый класс применяет этот метод для установки начальных значений атрибутов экземпляра и выполнения других задач начального запуска.

9. Двумя краеугольными камнями объектно-ориентированного кода Python являются специальный аргумент `self` в функциях методов и метод конструктора `__init__`; зная их, вы должны быть в состоянии читать большинство объектно-ориентированного кода на Python – помимо них это практически пакеты функций. Конечно, поиск в иерархии наследования тоже имеет значение, но `self` представляет автоматический объектный аргумент, а метод `__init__` широко распространен.

ГЛАВА 28

Более реалистичный пример

В следующей главе мы будем исследовать детали синтаксиса классов. Однако прежде чем заняться этим, имеет смысл рассмотреть пример работы с классами, более реалистичный, нежели то, что приводились до сих пор. Мы построим набор классов, делающих кое-что более конкретное – регистрацию и обработку сведений о людях. Вы увидите, что компоненты, которые в программировании на Python называются *экземплярами* и *классами*, часто способны исполнять такие же роли, как *записи* и *программы* в более традиционных терминах.

В частности мы планируем реализовать два класса:

- `Person` – класс, который создает и обрабатывает сведения о людях;
- `Manager` – настроенная версия класса `Person`, которая модифицирует унаследованное поведение.

Попутно мы создадим экземпляры обоих классов и протестируем их функциональность. Затем будет продемонстрирован подходящий сценарий использования для классов – мы сохраним наши экземпляры в объектно-ориентированной базе данных `shelve`, обеспечив их постоянство. В итоге вы сможете применять написанный код в качестве шаблона для формирования полноценной базы данных, реализованной полностью на Python.

Тем не менее, кроме реальной полезности настоящая глава также имеет и *учебный* характер: она предлагает руководство по ООП на Python. Люди часто схватывают основную идею синтаксиса классов, описанную в предыдущей главе, но им трудно понять, с чего начать, когда возникает необходимость в создании нового класса с нуля. С этой целью мы будем делать здесь по одному шагу за раз, чтобы помочь вам усвоить основы; классы будут строиться постепенно, так что вы сможете увидеть, как их функциональные средства объединяются в завершенные программы.

В конце концов, наши классы по-прежнему будут относительно небольшими по объему кода, но проиллюстрируют *все* основные идеи в модели ООП на языке Python. Несмотря на синтаксические детали, система классов Python в действительности сводится всего лишь к поиску атрибута в дереве объектов и к специальному первому аргументу в функциях.

Шаг 1: создание экземпляров

Итак, закончим стадию проектирования и приступим к реализации. Наша первая задача – начать написание кода главного класса, Person. Откроем текстовый редактор и создадим новый файл для кода, который будет написан. В Python принято довольно строгое соглашение начинать имена модулей с буквы нижнего регистра, а имена классов – с буквы верхнего регистра. Как и имя аргументов self в методах, язык этого не требует, но соглашение получило настолько широкое распространение, что отклонение от него может сбить с толку тех, кто впоследствии будет читать ваш код. Для соответствия соглашению мы назовем новый файл модуля person.py и назначим классу внутри него имя Person:

```
# Файл person.py (начало)

class Person:    # Начало класса
```

Вся работа внутри файла будет делаться далее в главе. В одном файле модуля Python можно создавать любое количество функций и классов, поэтому имя файла person.py может утратить смысл, если позже мы добавим в данный файл несвязанные компоненты. Пока что мы предположим, что абсолютно все в этом файле будет иметь отношение к Person. Вероятно, так и должно быть в любом случае – как выяснится, модули работают лучше всего, когда они преследуют единую цель *сцепления*.

Написание кода конструкторов

Первое, что мы хотим делать с помощью класса Person, связано с регистрацией основных сведений о людях – заполнением полей записей, если так понятнее. Разумеется, в терминологии Python они известны как *атрибуты* объекта экземпляра и обычно создаются путем присваивания значений атрибутам self в функциях методов класса. Нормальный способ предоставления атрибутам экземпляра первоначальных значений предусматривает их присваивание через self в *методе конструктора* `__init__`, который содержит код, автоматически выполняемый Python каждый раз, когда создается экземпляр. Давайте добавим к классу метод конструктора:

```
# Добавление инициализации полей записи

class Person:
    def __init__(self, name, job, pay): # Конструктор принимает три аргумента
        self.name = name                # Заполнить поля при создании
        self.job = job                  # self - новый объект экземпляра
        self.pay = pay
```

Такая схема написания кода весьма распространена: мы передаем данные, подлежащие присоединению к экземпляру, в виде аргументов методу конструктора и присваиваем их атрибутам self, чтобы сохранить их на постоянной основе. В терминах ООП аргумент self является вновь созданным объектом экземпляра, а name, job и pay становятся *информацией о состоянии* – описательными данными, сохраняемыми в объекте для использования в будущем. Хотя другие методики (такие как замыкания вложенных областей видимости) тоже способны сохранять детали, атрибуты экземпляра делают это очень явным и легким для понимания.

Обратите внимание, что имена аргументов здесь встречаются *дважды*. Поначалу код может даже показаться несколько избыточным, но это не так. Например, аргумент job представляет собой локальную переменную в области видимости функции `__init__`, но self.job – атрибут экземпляра, в котором передается подразумевае-

мый объект вызова метода. Они являются двумя разными переменными, по воле случая имеющие одно и то же имя. За счет присваивания локальной переменной job атрибуту self.job посредством self.job=job мы сохраняем переданное значение job в экземпляре для последующего применения. Как обычно в Python, предназначение имени определяется местом, где ему присваивается значение, или объектом, который ему присвоен.

Говоря об аргументах, с методом конструктора __init__ в действительности не связано ничего магического, помимо того факта, что он автоматически вызывается при создании экземпляра и имеет особым образом именованный первый аргумент. Несмотря на свое странное название, он представляет собой нормальную функцию и поддерживает все возможности функций, рассмотренные до сих пор. Скажем, мы можем указывать *стандартные значения* для ряда аргументов, так что их не придется предоставлять в случаях, когда они недоступны или бесполезны.

В целях демонстрации давайте сделаем аргумент job необязательным – он будет получать стандартное значение None, указывающее на то, что создаваемый экземпляр Person представляет человека, который (в текущий момент) не нанят на работу. Поскольку стандартным значением job будет None, тогда ради согласованности, вероятно, имеет смысл также установить стандартное значение для pay в 0 (если только какие-то ваши знакомые не умудряются получать заработную плату, не имея работы!). На самом деле мы обязаны указать стандартное значение для pay, т.к. в соответствии с правилами синтаксиса Python и главой 18 из первого тома все аргументы в заголовке функции, находящиеся после первого аргумента со стандартным значением, тоже должны иметь стандартные значения:

```
# Добавление стандартных значений для аргументов конструктора

class Person:
    def __init__(self, name, job=None, pay=0):    # Нормальные аргументы функции
        self.name = name
        self.job = job
        self.pay = pay
```

Такой код означает, что при создании экземпляров Person нам необходимо передавать имя, но аргументы job и pay теперь необязательны; они получат стандартные значения None и 0, когда опущены. Как обычно, аргумент self заполняется Python автоматически для ссылки на объект экземпляра – присваивание значений атрибутам self присоединяет их к новому экземпляру.

Тестирование в ходе дела

Пока что класс Person делает не особо многое (по существу он всего лишь заполняет поля новой записи), но является реальным рабочим классом. К настоящему моменту мы могли бы добавить код для дополнительных возможностей, но не будем этого делать. Как вы вероятно уже начали понимать, программирование на Python в действительности представляет собой вопрос *пошагового создания прототипов* – вы пишете какой-то код, тестируете его, пишете дополнительный код, снова тестируете и т.д. Поскольку Python обеспечивает нас как интерактивным сеансом, так и практически немедленным учетом изменений в коде, более естественно проводить тестирование в ходе дела, нежели писать крупный объем кода и тестировать его весь сразу.

Прежде чем добавлять дополнительные возможности, давайте протестируем то, что мы получили до сих пор, создав несколько экземпляров нашего класса и отобразив их атрибуты в том виде, как их присоединил конструктор. Мы могли бы делать это

интерактивно, но как вы уже наверняка догадались, интерактивному тестированию присущи свои ограничения — довольно утомительно заново импортировать модули и повторно набирать тестовые сценарии каждый раз, когда начинается новый сеанс тестирования. Чаще всего программисты на Python используют интерактивную подсказку для простых одноразовых тестов, но выполняют более существенное тестирование путем написания кода в конце файла, который содержит объекты, подлежащие тестированию:

```
# Добавление кода самотестирования

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

bob = Person('Bob Smith')                        # Тестирование класса
sue = Person('Sue Jones', job='dev', pay=100000)    # Автоматически
                                                    #   выполняет __init__
print(bob.name, bob.pay)                         # Извлечение присоединенных атрибутов
print(sue.name, sue.pay)                         # Атрибуты sue и bob отличаются
```

Обратите внимание, что объект bob принимает стандартные значения для job и pay, но объект sue предоставляет значения явно. Также взгляните на то, как применяются *ключевые аргументы* при создании sue. Мы могли бы взамен передавать аргументы по позиции, но ключевые аргументы могут помочь вспомнить назначение данных в более позднее время и позволяют передавать аргументы в любом желаемом порядке слева направо. Опять-таки, несмотря на необычное имя, `__init__` является нормальной функцией, поддерживающей все то, что вы уже знаете о функциях — в том числе стандартные значения и передаваемые по имени ключевые аргументы.

В случае запуска файла person.py как сценария тестовый код в конце файла создает два экземпляра нашего класса и выведет значения двух атрибутов каждого (name и pay):

```
C:\code> person.py
Bob Smith 0
Sue Jones 100000
```

Вы можете также набрать тестовый код данного файла в интерактивной подсказке Python (предварительно импортировав класс Person), но помещение кода заготовленных тестов внутрь файла модуля, как было показано выше, значительно облегчает их повторный запуск в будущем.

Хотя приведенный тестовый код довольно прост, он уже демонстрирует кое-что важное. Обратите внимание, что атрибут name объекта bob — не такой же, как у sue, а pay объекта sue — не такой же, как у bob. Каждый объект представляет собой независимую запись со сведениями. Формально bob и sue являются *объектами пространств имен* — подобно всем экземплярам классов каждый из них имеет собственную независимую копию информации о состоянии, созданную классом. Из-за того, что каждый экземпляр класса располагает своим набором атрибутов self, классы оказываются естественным инструментом для регистрации сведений для множества объектов. Как и встроенные типы вроде списков и словарей, классы служат своего рода *фабриками объектов*.

Другие программные структуры Python, такие как функции и модули, не поддерживают концепцию подобного рода. Функции замыканий из главы 17 первого тома близки с точки зрения сохранения состояния для каждого вызова, но не обладают множеством методов, наследованием и более крупной структурой, которые мы получает от классов.

Использование кода двумя способами

В том виде, как есть, тестовый код в конце файла работает, но есть большая загвоздка – его операторы `print` верхнего уровня выполняются и при запуске файла как сценария, и при импортировании как модуля. Это означает, что когда мы решим импортировать класс из файла `person.py`, чтобы применять его где-то в другом месте (и позже в главе мы так и поступим), то при каждом импортировании будем видеть вывод его тестового кода. Однако такая ситуация не может считаться подходящей для программного обеспечения: клиентские программы, скорее всего, не заботят наши внутренние тесты и в них нежелательно смешивать наш вывод с собственным выводом.

Хотя мы могли бы вынести тестовый код в отдельный файл, часто удобнее размещать код тестов в том же самом файле, где расположены тестируемые элементы. Было бы лучше организовать выполнение тестовых операторов в конце файла, *только* когда файл запускается для тестирования, а не когда он импортируется. Именно для этого предназначена проверка атрибута `__name__` модуля, как вы знаете из предыдущей части книги (находящейся в первом томе). Вот как выглядит такое дополнение:

```
# Дать возможность импортировать этот файл, а также запускать/тестировать
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__':     # Только когда запускается для тестирования
    # Код самотестирования
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

Теперь мы получаем в точности то поведение, к которому стремились – запуск файла как сценария верхнего уровня тестирует его, потому что `__name__` является `__main__`, но импортирование его в качестве библиотеки классов не приводит к выполнению тестов:

```
C:\code> person.py
Bob Smith 0
Sue Jones 100000

C:\code> python
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
>>> import person
>>>
```

При импортировании файл определяет класс, но не использует его. Когда файл запускается напрямую, он создает два экземпляра нашего класса, как и ранее, и выводит два атрибута каждого экземпляра; и снова из-за того, что каждый экземпляр представляет собой независимый объект пространства имен, значения их атрибутов отличаются.

Переносимость версий: print

Весь код в текущей главе работает в Python 2.X и 3.X, но я запускаю его под управлением Python 3.X, а для вывода применяю вызовы функции print с множеством аргументов из Python 3.X. Как объяснялось в главе 11 первого тома, это означает, что вывод может слегка варьироваться в случае запуска под управлением Python 2.X. Если вы запустите код в том виде, как есть, в Python 2.X, то он будет работать, но вы заметите круглые скобки в некоторых строках вывода, поскольку дополнительные круглые скобки в print из Python 2.X превращают элементы в кортеж:

```
C:\code> c:\python27\python person.py
('Bob Smith', 0)
('Sue Jones', 100000)
```

Если такое отличие оказывается той деталью, которая не дает вам покоя, тогда просто удалите круглые скобки, чтобы использовать оператор print из Python 2.X, или добавьте в начало сценария оператор импортирования функции print из Python 3.X, как делалось в главе 11 первого тома (я бы добавлял его повсюду, но слегка отвлекает):

```
from __future__ import print_function
```

Вы также можете избежать проблемы совместимости, связанной с круглыми скобками, за счет применения форматирования, чтобы выдавать одиночный объект, подлежащий выводу. Оба следующих оператора работают в Python 2.X и 3.X, хотя форма с методом новее:

```
print('{0} {1}'.format(bob.name, bob.pay))    # Метод форматирования
print('%s %s' % (bob.name, bob.pay))          # Выражение форматирования
```

Как объяснялось в главе 11 первого тома, в ряде случаев подобное форматирование оказывается обязательным, потому что объекты, *вложенные* в кортеж, могут выводиться не так, как при выводе в виде объектов верхнего уровня. Первые выводятся с помощью `__repr__`, а вторые посредством `__str__` (методы перегрузки операций, обсуждаемые далее в этой главе и в главе 30).

Чтобы обойти проблему, данная версия кода отображает с помощью `__repr__` (запасной вариант во всех случаях, включая вложение и интерактивную подсказку) вместо `__str__` (стандартный вариант для print), поэтому все появления объектов выводятся одинаково в Python 3.X и 2.X, даже те, что находятся в избыточных круглых скобках кортежей!

Шаг 2: добавление методов, реализующих поведение

Пока все выглядит хорошо — к настоящему моменту наш класс по существу является *фабрикой* записей; он создает и заполняет поля записей (атрибуты экземпляров, выражаясь терминами Python). Тем не менее, даже будучи настолько ограниченным классом, он позволяет выполнять некоторые операции над своими объектами. Несмотря на то что классы добавляют дополнительный уровень структуры, в конечном итоге они делают бóльшую часть своей работы, встраивая и обрабатывая *основные типы данных* наподобие списков и строк. Другими словами, если вам уже известно, как использовать простые основные типы Python, то вы уже знаете многое из истории о классах Python; классы в действительности представляют собой лишь незначительное структурное расширение.

Например, поле name в наших объектах – это простая строка, так что мы можем извлекать фамилии из объектов, разбивая по пробелам и индексируя. Все они являются операциями над основными типами данных, которые работают независимо от того, будут их объекты встроенными в экземпляры класса или нет:

```
>>> name = 'Bob Smith'    # Простая строка, за пределами класса
>>> name.split()          # Извлечение фамилии
['Bob', 'Smith']
>>> name.split()[-1]      # Или [1], если всегда есть только две части
'Smith'
```

Аналогичным образом мы можем повысить человеку заработную плату за счет обновления поля pay объекта, т.е. изменяя его информацию о состоянии на месте посредством присваивания. Такая задача также включает в себя базовые операции, которые работают с основными типами Python безотносительно к тому, автономны они или встроены в структуру класса (ниже применяется форматирование, чтобы скрыть тот факт, что разные версии Python выводят отличающееся количество десятичных цифр):

```
>>> pay = 100000             # Простая переменная, за пределами класса
>>> pay *= 1.10              # Предоставить повышение на 10%
>>> print('%.2f' % pay)      # Или pay = pay * 1.10, если вам нравится много набирать
110000.00                    # Или pay = pay + (pay * .10),
                             #   если это _действительно_ так!
```

Чтобы применить эти операции к объектам Person, созданным в сценарии, нужно просто сделать с bob.name и sue.pay то же самое, что мы делали с name и pay. Операции остаются теми же, но объекты присоединяются в качестве атрибутов к объектам, созданным из нашего класса:

```
# Обработка встроенных типов: строки, изменяемость

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])    # Извлечение фамилии из объекта
    sue.pay *= 1.10                # Предоставление этому объекту повышения
    print('%.2f' % sue.pay)
```

Здесь мы добавили последние три строки; когда они выполняются, производится извлечение фамилии из объекта bob с использованием базовых строковых и списковых операций в отношении его поля name и повышение заработной платы объекту sue за счет модификации атрибута pay на месте с помощью базовых числовых операций. В определенном смысле sue также является *изменяемым* объектом – его состояние модифицируется на месте почти как список после вызова append. Ниже приведен вывод новой версии:

```
Bob Smith 0
Sue Jones 100000
Smith
110000.00
```

Предыдущий код работает, как было запланировано, но если вы покажете его опытному разработчику программного обеспечения, то он, скорее всего, сообщит вам, что такой универсальный подход – не особо хорошая идея для воплощения на практике. Жесткое кодирование операций вроде этих *за пределами* класса может привести к проблемам с сопровождением в будущем.

Скажем, что если вы жестко закодируете способ извлечения фамилии во многих местах программы? Когда его понадобится изменить (например, для поддержки новой структуры имени), то вам придется отыскать и обновить *каждое* вхождение. Подобным же образом, если изменится код повышения заработной платы (например, чтобы требовать одобрения или обновлений базы данных), то у вас может быть множество копий, подлежащих модификации. Один лишь поиск всех вхождений такого кода в крупных программах может оказаться проблематичным – они могут быть разбросаны по многим файлам, разделены на индивидуальные шаги и т.д. В прототипе подобного рода частые изменения почти гарантированы.

Написание кода методов

На самом деле мы здесь хотим задействовать концепцию проектирования программного обеспечения, известную как *инкапсуляция* – помещение операционной логики в оболочку интерфейсов, чтобы код каждой операции был написан только один раз в программе. Тогда если в будущем возникнет необходимость в изменении, то изменять нужно будет только одну копию. Более того, мы можем практически произвольно изменять внутренности одиночной копии, не нарушая работу кода, который ее потребляет.

Выражаясь терминами Python, мы хотим поместить код операций над объектами в *методы* класса, а не засорять ими всю программу. Фактически это одно из дел, с которыми классы справляются очень хорошо – *вынесение* кода с целью устранения *избыточности* и в итоге повышения удобства сопровождения. В качестве дополнительного бонуса помещение операций внутрь методов позволяет применять их к любому экземпляру класса, а не только к тем, где они были жестко закодированы для обработки.

На практике все описанное выглядит проще, чем в теории. В следующем коде инкапсуляция достигается переносом двух операций из кода за пределами класса в методы внутри класса. Далее изменим код самотестирования в конце файла, чтобы использовать в нем новые методы вместо жестко закодированных операций:

```
# Добавление методов для инкапсуляции операций с целью повышения удобства
сопровождения

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):                          # Методы реализации поведения
        return self.name.split()[-1]             # self - подразумеваемый объект
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))    # Потребуется изменять
                                                     #   только здесь

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

```
print(bob.lastName(), sue.lastName())     # Использовать новые методы
sue.giveRaise(.10)                        # вместо жесткого кодирования
print(sue.pay)
```

Как уже известно, *методы* представляют собой нормальные функции, которые присоединяются к классам и предназначены для обработки экземпляров этих классов. Экземпляр является подразумеваемым объектом вызова метода и передается в аргументе self автоматически.

Трансформация в методы в данной версии прямолинейна. Например, новый метод lastName просто делает в отношении self то, что в предыдущей версии было жестко закодировано для bob, поскольку self — подразумеваемый объект при вызове метода. Метод lastName также возвращает результат, потому что теперь эта операция представляет собой вызываемую функцию; он вычисляет значение для произвольного применения в вызывающем коде, даже когда оно всего лишь выводится. Аналогично новый метод giveRaise просто делает с self то, что раньше выполнялось с sue.

Запуск файла приводит к получению такого же вывода, как ранее — мы в основном лишь провели *рефакторинг* кода, чтобы облегчить его модификацию в будущем, а не изменили поведение:

```
Bob Smith 0
Sue Jones 100000
Smith Jones
110000
```

Здесь стоит пояснить несколько деталей, касающихся кода. Во-первых, обратите внимание, что хранящий величину заработной платы атрибут pay в sue по-прежнему остается целым числом после поднятия — мы преобразуем математический результат в целое число, вызывая внутри метода встроенную функцию int. Изменение типа значения на int или float возможно не особо существенная проблема в рассматриваемой демонстрации: объекты целых чисел и чисел с плавающей точкой имеют те же самые интерфейсы и могут смешиваться внутри выражений. Однако в реальной системе нам может понадобиться решить проблемы с усечением и округлением — деньги для экземпляров Person наверняка важны!

Как известно из главы 5 первого тома, мы могли бы справиться с этим, используя вызов встроенной функции round(N, 2) для округления и оставления центов, применяя тип decimal с фиксированной точностью либо сохраняя денежные значения как полные числа с плавающей точкой и отображая их с использованием строки формата %.2f или {0:.2f}, которая обеспечивает вывод центов, как делалось ранее. Пока что мы просто усекаем любые центы посредством int. Другая идея была воплощена в функции money из модуля formats.py в главе 25; можете импортировать указанный модуль, чтобы отображать величину заработной платы с запятыми, центами и символами валюты.

Во-вторых, обратите внимание на то, что на этот раз мы также выводим фамилию объекта sue — поскольку логика извлечения фамилии была инкапсулирована в методе, мы можем применять его к *любому экземпляру* класса. Как мы видели, Python сообщает методу, какой экземпляр обрабатывать, автоматически передавая его в первом аргументе, который обычно называется self. В частности:

- в первом вызове, bob.lastName(), bob является подразумеваемым объектом, передаваемым self;
- во втором вызове, sue.lastName(), в self передается sue.

Отследите указанные вызовы, чтобы посмотреть, каким образом экземпляр оказывается в `self` — это ключевая концепция. Совокупный эффект в том, что метод каждый раз извлекает фамилию из подразумеваемого объекта. То же самое происходит с методом `giveRaise`. Скажем, мы могли бы предоставить объекту `bob` повышение, вызывая аналогичным образом метод `giveRaise` для обоих экземпляров. Тем не менее, к сожалению, его нулевое начальное значение заработной платы воспрепятствует получению повышения при текущей реализации программы — умножение нуля на что угодно даст ноль, и вполне возможно, мы захотим решить эту проблему в будущем выпуске нашей программы.

Наконец, обратите внимание, что метод `giveRaise` полагается на передачу в аргументе `percent` числа с плавающей точкой между нулем и единицей. В реальности такое допущение может быть излишне радикальным (повышение на 1000% вероятно для большинства из нас было бы воспринято как ошибка!); в прототипе мы оставим его, как есть, но в будущем выпуске возможно понадобится организовать проверку или хотя бы документировать такую особенность. Позже в книге идея будет изложена другими словами, когда мы займемся так называемыми декораторами функций и оператором `assert` языка Python — альтернативы, которые могут выполнять проверки достоверности автоматически на стадии разработки. Например, в главе 39 мы реализуем инструмент, который позволит проверять достоверность с помощью удивительных магических формул вроде показанных ниже:

```
@rangetest(percent=(0.0, 1.0))     # Использование декоратора для проверки
                                   #   достоверности
def giveRaise(self, percent):
    self.pay = int(self.pay * (1 + percent))
```

Шаг 3: перегрузка операций

В данный момент мы располагаем довольно многофункциональным классом, который генерирует и инициализирует экземпляры, а также поддерживает две новые линии поведения для обработки экземпляров в форме методов. Пока все в порядке.

Однако в нынешнем виде тестирование все еще менее удобно, чем должно быть — для трассировки объектов нам приходится вручную извлекать и вводить *индивидуальные атрибуты* (скажем, `bob.name`, `sue.pay`). Было бы хорошо, если бы отображение экземпляра всего сразу действительно давало какую-то полезную информацию. К сожалению, стандартный формат отображения для объекта экземпляра не особо подходит — он выводит имя класса объекта и его адрес в памяти (что в Python по существу бесполезно за исключением уникального идентификатора).

Чтобы увидеть это, изменим последнюю строку в сценарии на `print(sue)`, обеспечив отображение объекта как единого целого. Ниже приведено то, что мы получим — вывод указывает на то, что `sue` является объектом (`object`) в Python 3.X и экземпляром (`instance`) в Python 2.X:

```
Bob Smith 0
Sue Jones 100000
Smith Jones
<__main__.Person object at 0x00000000029A0668>
```

Реализация отображения

К счастью, положение дел легко улучшить, задействовав *перегрузку операций* — написать код методов в классе, которые перехватывают и обрабатывают встроенные

операции, когда выполняются на экземплярах класса. В частности, мы можем использовать вторые по частоте применения в Python методы перегрузки операций после `__init__`: метод `__repr__`, который мы реализуем здесь, и его двойник `__str__`, представленный в предыдущей главе.

Методы выполняются автоматически каждый раз, когда экземпляр преобразуется в свою строку вывода. Поскольку именно это происходит при выводе объекта, в результате вывод объекта отображает то, что возвращается методом `__str__` или `__repr__` объекта, если объект либо самостоятельно определяет такой метод, либо наследует его от суперкласса. Имена с двумя символами подчеркиваниями наследуются подобно любым другим.

Формально `__str__` предпочтительнее `print` и `str`, а `__repr__` используется в качестве запасного варианта для данных ролей и во всех остальных контекстах. Хотя можно применить два метода для реализации отличающегося отображения в разных контекстах, написание кода одного лишь `__repr__` достаточно, чтобы предоставить единственное отображение во всех случаях — вывод с помощью `print`, вложенные появления и эхо-вывод в интерактивной подсказке. Клиенты по-прежнему располагают возможностью предоставления альтернативного отображения посредством `__str__`, но только для ограниченных контекстов; так как рассматриваемый пример изолирован, вопрос здесь оказывается спорным.

Метод конструктора `__init__`, код которого мы уже написали, строго говоря, тоже является перегрузкой операции — он выполняется автоматически во время создания для инициализации нового экземпляра. Тем не менее, конструкторы настолько распространены, что они не выглядят похожими на особый случай. Более специализированные методы вроде `__repr__` позволяют подключаться к специфическим операциям и обеспечивать *специальное поведение*, когда объекты используются в таких контекстах.

Давайте напишем код. Ниже наш класс расширяется, чтобы предоставляет специальное отображение со списком атрибутов при выводе экземпляров класса как единого целого, не полагаясь на менее полезное стандартное отображение:

```
# Добавление метода перегрузки операции __repr__ для вывода объектов

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):                                          # Добавленный метод
        return '[Person: %s, %s]' % (self.name, self.pay) # Строка для вывода

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

Обратите внимание, что в `__repr__` для построения строки, подлежащей отображению, применяется операция `%` строкового форматирования; для выполнения своей работы классы используют объекты и операции встроенных типов, как здесь показано. И снова все, что вам уже известно о встроенных типах и функциях, применимо к коду, основанному на классах. По большому счету классы лишь добавляют дополнительный уровень *структуры*, которая упаковывает функции и данные вместе и поддерживает расширение.

Мы также изменили код самотестирования для отображения объектов напрямую вместо вывода индивидуальных атрибутов. Получаемый в результате запуска вывод теперь оказывается более ясным и понятным; строки `[...]`, возвращаемые новым методом `__repr__`, выполняются автоматически операциями вывода:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
```

Примечание по проектному решению: как будет показано в главе 30, метод `__repr__`, если присутствует, то часто используется, чтобы предоставить низкоуровневое отображение объекта как в коде, а метод `__str__` зарезервирован для более информативного отображения, дружественного к пользователям. Иногда классы предлагают и `__str__` для дружественного к пользователям отображения, и `__repr__` с добавочными деталями для просмотра разработчиками. Поскольку операция вывода запускает `__str__`, а интерактивная подсказка автоматически выводит результаты с помощью `__repr__`, это можно применять для снабжения обеих целевых аудиторий подходящим отображением.

Так как метод `__repr__` пригоден в большем количестве случаев отображения, в том числе при вложенных появлениях, а нас не интересует отображение двух разных форматов, то для класса `Person` вполне достаточно реализации всеобъемлющего метода `__repr__`. Здесь это также означает, что наше специальное отображение будет использоваться в Python 2.X в случае указания `bob` и `sue` в вызове `print` из Python 3.X – формально вложенное появление, согласно врезке "Переносимость версий: `print`" ранее в главе.

Шаг 4: настройка поведения за счет создания подклассов

На текущем этапе наш класс реализует бо́льшую часть механизма ООП в Python: он создает экземпляры, снабжает поведением в методах и даже немного перегружает операции, чтобы перехватывать операции вывода в `__repr__`. Он фактически упаковывает наши данные и логику вместе в единственный автономный *программный компонент*, облегчая нахождение кода и его изменение в будущем. Разрешая нам инкапсулировать поведение, он также дает возможность выносить такой код во избежание избыточности и связанных затруднений при сопровождении.

Не охваченной осталась только одна значительная концепция ООП – *настройка через наследование*. В некотором смысле мы уже привлекали к делу наследование, потому что экземпляры наследуют методы от своих классов. Однако для демонстрации реальной мощи ООП необходимо определить отношение суперкласс/подкласс, которое позволит расширять наше программное обеспечения и замещать унаследованные линии поведения. В конце концов, это главная идея ООП; поощряя кодовую модель,

основанную на настройке уже сделанной работы, ООП значительно сокращает время разработки.

Написание кода подклассов

В качестве следующего шага давайте задействуем методологию ООП для применения и настройки нашего класса Person, расширив имеющуюся программную иерархию. В целях рассматриваемого учебного пособия мы определим подкласс класса Person по имени Manager, который замещает унаследованный метод giveRaise более специализированной версией. Новый класс начинается так:

```
class Manager(Person):     # Определение подкласса Person
```

Код означает, что мы определяем новый класс по имени Manager, который унаследован от суперкласса Person и может добавлять настройки. Говоря проще, класс Manager очень похож на Person, но Manager располагает специальным способом получения повышений.

В плане аргумента давайте предположим, что когда экземпляр Manager получает повышение, он принимает переданный процент обычным образом, но также становится обладателем добавочной премии, имеющей стандартное значение 10%. Например, если повышение для экземпляра Manager указано как 10%, то в действительности он получит 20%. (Разумеется, любые совпадения с реальными людьми совершенно случайны.) Новый метод начинается, как показано ниже; из-за того, что в дереве классов это переопределение giveRaise будет находиться ближе к экземплярам Manager, чем исходная версия в Person, оно фактически замещает и таким образом настраивает операцию. Вспомните, что в соответствии с правилами поиска в иерархии наследования выигрывает версия имени, расположенная ниже всех[1]:

```
class Manager(Person):                          # Наследование атрибутов Person
    def giveRaise(self, percent, bonus=.10):  # Переопределение с целью настройки
```

Расширение методов: плохой способ

Существуют два способа написания кода для такой настройки класса Manager: хороший и плохой. Мы начнем с *плохого способа*, т.к. он может быть чуть легче для понимания. Плохой способ заключается в том, чтобы вырезать код giveRaise из класса Person и вставить его в класс Manager, после чего модифицировать:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus)) # Плохой способ:
                                                          # вырезание и вставка
```

Все работает, как заявлено — когда мы позже вызываем метод giveRaise экземпляра Manager, будет выполнена созданная специальная версия, начисляющая добавочную премию. Что же тогда не так с кодом, который выполняется корректно?

Проблема здесь имеет очень общий характер: всякий раз, когда вы копируете код посредством вырезания и вставки, то по существу *удваиваете* объем работ по сопровождению в будущем. Подумайте об этом: поскольку мы копируем первоначальную версию, если когда-либо понадобится изменить способы выдачи повышений (что наверняка произойдет), то нам придется модифицировать код в *двух* местах, а не в одном.

---

[1] Конечно, не в обиду любым менеджерам из читательской аудитории. Когда-то я вел обучающий курс по Python в Нью-Джерси и, между прочим, никто не смеялся над этой шуткой. Позже организаторы сказали мне, что группа состояла из менеджеров, оценивающих Python.

Несмотря на то что пример является простым и искусственным, он демонстрирует универсальную проблему – всякий раз, когда возникает искушение программировать путем копирования кода, вероятно, стоит поискать более подходящий подход.

Расширение методов: хороший способ

Что мы действительно хотим здесь сделать – каким-то образом *расширить* исходный метод giveRaise вместо его полного замещения. *Хороший способ* в Python предусматривает вызов исходной версии напрямую с дополненными аргументами:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus) # Хороший способ: расширение
                                                #   исходной версии
```

В коде задействован тот факт, что метод класса всегда может быть вызван либо через *экземпляр* (обычный способ, когда Python автоматически передает экземпляр аргументу self), либо через *класс* (менее распространенная схема, при которой экземпляр должен передаваться вручную). Если более конкретно, то вспомните, что нормальный вызов метода следующего вида:

```
экземпляр.метод(аргументы...)
```

автоматически транслируется Python в такую эквивалентную форму:

```
класс.метод(экземпляр, аргументы...)
```

где класс, который содержит подлежащий выполнению метод, определяется правилами поиска в иерархии наследования, применяемыми к имени метода. Вы можете использовать в своем сценарии *любую* из двух форм, но между ними наблюдается легкая асимметрия – вы обязаны помнить о необходимости передачи экземпляра вручную, если вызывает метод напрямую через класс. Так или иначе, но метод нуждается в объекте экземпляра, и Python предоставляет его автоматически только для вызовов, сделанных через экземпляр. В случае вызова через имя класса вы должны самостоятельно передавать экземпляр атрибуту self; для кода внутри метода вроде giveRaise аргумент self уже *является* объектом, на котором произведен вызов, и отсюда экземпляром, подлежащим передаче.

Прямой вызов через класс фактически отменяет поиск в иерархии наследования и запускает вызов выше в дереве классов, чтобы выполнить специфическую версию. В нашем случае мы можем применять такую методику для обращения к стандартной версии метода giveRaise из Person, несмотря на то, что он переопределен на уровне класса Manager. В ряде случаев мы просто *обязаны* вызывать через класс Person, т.к. вызов self.giveRaise() внутри кода giveRaise в Manager привел бы к зацикливанию. Дело в том, что self уже представляет собой экземпляр Manager, т.е. вызов self.giveRaise() был бы распознан снова как Manager.giveRaise и так далее *рекурсивно* до тех пор, пока не исчерпается доступная память.

"Хорошая" версия может выглядеть мало отличающейся в плане кода, но она способна значительно изменить будущее *сопровождение кода* – так как теперь логика giveRaise расположена лишь в одном месте (метод класса Person), у нас есть только одна версия, подлежащая модификации в будущем, если в том возникнет необходимость. И действительно, эта форма в любом случае отражает наше намерение более прямо – мы хотим выполнить стандартную операцию giveRaise, но просто добавить дополнительную премию. Ниже приведен полный код модуля с примененным последним шагом:

```
# Добавление настройки поведения в подкласс
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def giveRaise(self, percent, bonus=.10):   # Переопределить на этом уровне
        Person.giveRaise(self, percent + bonus) # Вызвать версию из Person

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)     # Создать экземпляр
Manager: __init__
    tom.giveRaise(.10)                      # Выполняется специальная версия
    print(tom.lastName())                   # Выполняется унаследованный метод
    print(tom)                              # Выполняется унаследованный __repr__
```

Чтобы протестировать настройку Manager, мы также добавили код самотестирования, который создает экземпляр Manager, вызывает его методы и выводит сам экземпляр. При создании экземпляра Manager мы передаем имя, а также необязательную должность и размер заработной платы — поскольку конструктор __init__ в классе Manager отсутствует, он наследует его от Person. Вот вывод новой версии:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

Здесь все выглядит хорошо: экземпляры bob и sue такие же, как прежде, а когда экземпляр Manager по имени tom получает повышение на 10%, то на самом деле становится обладателем 20% (его оплата возрастает с 50 000 до 60 000), потому что настроенная версия giveRaise из Manager выполняется только для него. Также обратите внимание, что вывод экземпляра tom как единого целого в конце тестового кода отображает элегантный формат, определенный в методе __repr__ класса Person: объекты Manager получают код __repr__, lastName и метода конструктора __init__ "бесплатно" от Person благодаря наследованию.

Как насчет super?

Для расширения унаследованных методов в примерах данной главы вызывались исходные методы через имя суперкласса: `Person.giveRaise(...)`. Это традиционная и простейшая схема в Python, которая используется в большей части книги.

Возможно, программистам на Java будет особенно интересно узнать, что в Python также имеется встроенная функция `super`, которая позволяет вызывать методы суперкласса более обобщенно. Тем не менее, в Python 2.X применять ее громоздко; она отличается по форме в линейках Python 2.X и Python 3.X; полагается на необычную семантику в Python 3.X; шероховато работает с перегрузкой операций в Python; и не всегда хорошо сочетается с традиционно реализованным множественным наследованием, где обращения к единственному суперклассу будет недостаточно.

В защиту вызова `super` следует отметить, что с ним тоже связан допустимый сценарий использования (совместная координация одинаково именованных методов в деревьях множественного наследования). Однако он опирается на упорядочение классов MRO (Method Resolution Order — порядок распознавания методов), которое многие считают экзотическим и искусственным; нереалистично предполагает, что будет надежно применяться универсальное развертывание; не полностью поддерживает замещение методов и списки аргументов переменной длины; к тому же для многих использование сценария, редко встречающегося в реальном коде на Python, представляется невразумительным решением.

Из-за перечисленных недостатков предпочтение в книге отдается обращению к суперклассам по явным именам, а не посредством `super`, та же политика рекомендуется для новичков и представление встроенной функции `super` откладывается до главы 32. О встроенной функции `super` обычно лучше судить после того, как вы освоите более простые и в целом более традиционные, в стиле Python, способы достижения тех же целей, особенно если вы — новичок в ООП. Темы вроде MRO и совместной координации методов в деревьях множественного наследования одинаково именованных методов не кажутся слишком востребованными начинающими, впрочем, как и остальными.

Мой совет всем программистам на Java, входящим в читательскую аудиторию: я предлагаю не поддаваться искушению и не применять встроенной функции `super` из Python до тех пор, пока у вас не появится возможность изучить связанные с ней тонкие последствия. Как только вы перейдете на множественное наследование, она окажется не тем, что вы думаете, а чем-то большим, нежели ваши возможные ожидания. Вызываемый класс может вообще не быть суперклассом и даже варьироваться в зависимости от контекста. Или переформулировав фразу из фильма "Форрест Гамп": встроенная функция `super` из Python как коробка шоколадных конфет — *никогда не знаешь, какая начинка тебе попадется!*

Полиморфизм в действии

Чтобы сделать приобретение унаследованного поведения еще более удивительным, мы можем временно добавить в конец файла следующий код:

```
if __name__ == '__main__':
    ...
    print('--All three--')
    for obj in (bob, sue, tom):       # Обработать объекты обобщенным образом
        obj.giveRaise(.10)            # Выполнить метод giveRaise этого объекта
        print(obj)                    # Выполнить общий метод __repr__
```

Вот какой в результате получается вывод (новые строки выделены полужирным):

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
--All three--
[Person: Bob Smith, 0]
[Person: Sue Jones, 121000]
[Person: Tom Jones, 72000]
```

В добавленном коде obj является экземпляром либо Person, либо Manager, и Python выполняет соответствующий метод giveRaise автоматически – нашу исходную версию в классе Person для bob и sue и настроенную версию в Manager для tom. Отследите вызовы методов самостоятельно, чтобы увидеть, каким образом Python выбирает правильный метод giveRaise для каждого объекта.

Это всего лишь демонстрация в работе понятия *полиморфизма* в Python, с которым мы встречались ранее в книге – то, что делает метод giveRaise, зависит от того, на чем он вызывается. Все становится более очевидным, когда средство полиморфизма выбирает среди написанного нами кода в классах. Практический эффект данного кода заключается в том, что экземпляр sue получает дополнительно 10%, но экземпляр tom – 20%, потому что выбор метода giveRaise координируется на основе типа объекта. Как уже известно, полиморфизм является центральной частью гибкости Python. Скажем, передача любого из трех объектов функции, которая вызывает метод giveRaise, дала бы такой же результат: в зависимости от типа переданного объекта автоматически выполнилась бы подходящая версия.

С другой стороны, вывод выполняет *тот же самый* метод __repr__ для всех трех объектов, т.к. он реализован только один раз в Person. Класс Manager специализирует и применяет код, который мы первоначально написали в Person. Несмотря на небольшой размер примера, он уже задействует способности ООП для настройки и многократного использования кода; благодаря классам это порой кажется почти автоматическим.

Наследование, настройка и расширение

На самом деле классы могут быть даже более гибкими, чем вытекает из рассмотренного примера. В общем случае классы способны наследовать, настраивать или расширять существующий код в суперклассах. Например, хотя внимание здесь сосредоточено на настройке, мы также добавляем в класс Manager уникальные методы, которые отсутствуют в Person, если экземпляры Manager требуют чего-то совершенно другого (тут снова на ум приходит фильм "Монти Пайтон: а теперь нечто совсем другое"). Сказанное иллюстрируется в следующем коде, где giveRaise переопределяет метод суперкласса с целью его настройки, но someThingElse определяет кое-что новое для расширения:

```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __repr__(self): ...

class Manager(Person):                    # Наследование
    def giveRaise(self, ...): ...         # Настройка
    def someThingElse(self, ...): ...     # Расширение
```

```
tom = Manager()
tom.lastName()                          # Буквальное наследование
tom.giveRaise()                         # Настроенная версия
tom.someThingElse()                     # Метод расширения
print(tom)                              # Унаследованный перегруженный метод
```

Дополнительные методы наподобие someThingElse в показанном коде *расширяют* существующее программное обеспечение и доступны только для объектов Manager, но не Person. Тем не менее, ради целей этого обучающего руководства мы ограничимся настройкой части поведения класса Person путем его переопределения, не добавляя новые линии поведения.

Объектно-ориентированное программирование: основная идея

В том виде, как есть, код может быть небольшим, но достаточно функциональным. И действительно, он демонстрирует основную идею, лежащую в основе ООП в целом: мы программируем путем настройки того, что уже было сделано, а не копирования либо изменения существующего кода. На первый взгляд это не всегда очевидный выигрыш, особенно с учетом требований по написанию добавочного кода классов. Но, в общем, стиль программирования, подразумеваемый классами, может радикально сократить время разработки по сравнению с другими подходами.

Скажем, в нашем примере теоретически мы могли бы реализовать специальную операцию giveRaise, не создавая подкласс, но никакой другой вариант не обеспечил бы получение настолько оптимального кода.

- Несмотря на то что мы могли бы просто реализовать класс Manager с нуля как новый, независимый код, нам пришлось бы повторно реализовать все линии поведения из Person, которые остались такими же в Manager.
- Хотя мы могли бы просто изменить существующий код класса Person на месте для удовлетворения требований операции giveRaise из Manager, это вероятно нарушило бы работу кода в местах, где по-прежнему необходимо исходное поведение Person.
- Несмотря на то что мы могли бы просто скопировать класс Person полностью, переименовать копию на Manager и изменить код операции giveRaise, это ввело бы избыточность кода, приводя к удваиванию работы по сопровождению — изменения, вносимые в будущем в класс Person, не будут подхватываться автоматически, и их придется вручную распространять на код Manager. Как обычно, подход с вырезанием и вставкой в текущий момент может выглядеть быстрым, но он удваивает объем работы в будущем.

Настраиваемые иерархии, которые мы можем строить с помощью классов, обеспечивают гораздо лучшее решение для программного обеспечения, развивающегося с течением времени. Другие инструменты в Python такой режим разработки не поддерживают. Располагая возможностью подгонки и расширения результатов выполненной ранее работы за счет реализации новых подклассов, мы можем задействовать то, что уже готово, а не начинать каждый раз с нуля, нарушать работу имеющегося кода или вводить множество копий кода, которые вероятно придется обновлять в будущем. При надлежащем применении ООП является мощным союзником программиста.

Шаг 5: настройка конструкторов

Код работает в том виде, как есть, но если вы более внимательно исследуете текущую версию, то столкнетесь с небольшой странностью – указание названия должности mgr для объектов Manager при их создании кажется бессмысленным: это вытекает из самого класса. Было бы лучше иметь возможность каким-то образом заполнять данное значение автоматически, когда создается экземпляр Manager.

Здесь мы можем использовать *тот же самый* трюк, который был задействован в предыдущем разделе: мы хотим настроить логику конструктора для объектов Manager таким образом, чтобы предоставлять название должности автоматически. В переводе на код нам нужно переопределить метод __init__ в классе Manager с целью предоставления строки mgr. Как и в настройке giveRaise, нам также необходимо выполнять исходный метод __init__ из Person, вызывая его через имя класса, чтобы он по-прежнему инициализировал атрибуты информации о состоянии наших объектов.

Задачу решает следующее расширение person.py – мы написали код нового конструктора Manager и изменили вызов, который создает экземпляр tom, убрав из него передачу названия должности mgr:

```
# Файл person.py
# Добавление настройки конструктора в подклассе

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def __init__(self, name, pay):                    # Переопределить конструктор
        Person.__init__(self, name, 'mgr', pay)  # Выполнить исходный с 'mgr'
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)    # Название должности не требуется:
    tom.giveRaise(.10)                   # Оно подразумевается/
                                         #   устанавливается классом
    print(tom.lastName())
    print(tom)
```

При расширении конструктора __init__ мы снова применяем ту же самую методику, которую ранее использовали для giveRaise – выполняем версию из суперкласса

путем вызова через имя класса напрямую и явно передаем экземпляр self. Хотя конструктор имеет странное имя, эффект идентичен. Поскольку нужно также выполнить логику создания экземпляра Person (для инициализации атрибутов экземпляра), мы действительно обязаны вызывать ее таким способом; иначе экземпляры не получат присоединенных атрибутов.

Подобного рода вызов конструкторов суперклассов из переопределенных версий оказывается весьма распространенным стилем программирования в Python. Сам по себе Python применяет наследование для поиска и вызова только *одного* метода `__init__` на стадии конструирования – расположенного *ниже всех* в дереве классов. Если необходимо, чтобы на стадии конструирования выполнялись методы `__init__`, находящиеся выше в дереве классов (обычно так и есть), тогда вы должны вызывать их вручную, как правило, через имя суперкласса. Положительный аспект здесь в том, что вы можете явно указывать аргумент для передачи конструктору суперкласса или даже вообще *не* вызывать его: отказ от вызова конструктора суперкласса позволяет вместо расширения полностью замещать его логику.

Вывод кода самотестирования этого файла такой же, как ранее – мы не изменяли то, что он делает, а просто реструктурировали, избавившись от некоторой логической избыточности:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

Объектно-ориентированное программирование проще, чем может казаться

Несмотря на свои относительно небольшие размеры, в такой полной форме наши классы охватывают почти все важные концепции механизма ООП в Python:

- создание экземпляров – заполнение атрибутов экземпляров;
- методы, реализующие поведение – инкапсуляция логики в методах класса;
- перегрузка операций – обеспечение линии поведения для встроенных операций вроде вывода;
- настройка поведения – переопределение методов в подклассах для их специализации;
- настройка конструкторов – добавление логики инициализации к шагам суперкласса.

Большинство перечисленных концепций основаны всего лишь на трех простых идеях: поиск атрибутов в иерархии наследования в форме деревьев объектов, особый аргумент self в методах и автоматическое сопоставление перегруженных операций с методами.

Попутно мы также сделали наш код легким для изменения в будущем, используя предрасположенность класса к вынесению кода с целью сокращения *избыточности*. Скажем, мы поместили логику внутрь методов и вызывали методы суперкласса из расширений, чтобы избежать наличия множества копий того же самого кода. Большинство таких шагов были естественным ростом структурной мощи классов.

В общем и целом это все, что есть в Python для ООП. Безусловно, классы могут становиться крупнее, чем было здесь продемонстрировано, и существуют более сложные концепции, связанные с классами, такие как декораторы и метаклассы, которые будут обсуждаться в последующих главах. Однако с точки зрения основ наши классы уже воплощают все упомянутые ранее концепции. На самом деле, если вы поняли работу написанных нами классов, тогда большая часть объектно-ориентированного кода на Python не должна вызывать особых вопросов.

Другие способы комбинирования классов

Сказав это, я обязан также сообщить вам о том, что хотя базовый механизм ООП в Python прост, способы объединения классов вместе в крупных программах – в определенной мере искусство. В настоящем руководстве мы концентрируемся на наследовании, потому что оно представляет собой механизм, поддерживаемый языком Python, но программисты иногда комбинируют классы и другими способами.

Например, распространенный стиль программирования предусматривает вложение объектов друг в друга для построения *составных объектов*. Мы детально исследуем такой стиль в главе 31, где речь пойдет больше о проектировании, чем о самом языке Python. Тем не менее, в качестве короткого примера мы могли бы применить эту идею комбинирования для реализации расширения `Manager`, *внедряя* класс `Person`, а не наследуя от него.

Показанная ниже альтернатива, находящаяся в файле `person-composite.py`, реализует такой прием за счет использования метода для перегрузки операции `__getattr__`, чтобы перехватывать извлечения неопределенных атрибутов и делегировать выполнение работы внедренному объекту посредством встроенной функции `getattr`. Вызов `getattr` был представлен в главе 25 первого тома (он аналогичен записи извлечения атрибута `X.Y` и потому приводит к поиску в иерархии наследования, но имя атрибута `Y` является строкой времени выполнения), а метод `__getattr__` будет полностью раскрыт в главе 30, однако здесь вполне достаточно его базового применения.

Путем комбинирования указанных инструментов метод `giveRaise` по-прежнему обеспечивает настройку, изменяя аргумент, который передается внедренному объекту. В действительности `Manager` становится уровнем контроллера, передающим вызовы *вниз* внедренному объекту, а не *вверх* методам суперкласса:

```
# Файл person-composite.py
# Альтернативная версия Manager, основанная на внедрении

class Person:
    ... тот же код, что и ранее...

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)    # Внедрить объект Person
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)    # Перехватить и делегировать
    def __getattr__(self, attr):
        return getattr(self.person, attr)  # Делегировать все остальные атрибуты
    def __repr__(self):
        return str(self.person)            # Снова должен быть перегружен
                                           #   (в Python 3.X)

if __name__ == '__main__':
    ...тот же код, что и ранее...
```

Вывод новой версии будет таким же, как у предыдущей версии, так что нет смысла приводить его повторно. Более важно здесь то, что данная альтернативная версия Manager иллюстрирует общий стиль программирования, обычно известный как *делегирование* – структура на основе составного объекта, которая управляет внедренным объектом и передает ему вызовы методов.

Шаблон удалось внедрить в нашем примере, но он потребовал почти в два раза больше кода и он не настолько хорошо подходит для задействованных видов настроек, как наследование (на самом деле, ни один здравомыслящий программист на Python никогда бы не реализовывал приведенный пример подобным образом на практике, разве что при написании учебного пособия!). Здесь объект Manager не является Person, поэтому нам необходим добавочный код для ручной отправки вызовов методов внедренному объекту. Методы перегрузки операций вроде __repr__ должны быть переопределены (по крайней мере, в Python 3.X, как будет отмечено во врезке "Перехват встроенных атрибутов в Python 3.X" далее в главе). Кроме того, добавление нового поведения в Manager менее прямолинейно, поскольку информация о состоянии смещена на один уровень ниже.

Тем не менее, *внедрение объектов* и основанные на нем паттерны проектирования могут очень хорошо подходить, когда внедренные объекты требуют более ограниченного взаимодействия с контейнером, чем подразумевает прямая настройка. Скажем, уровень контроллера, или *посредника*, подобный альтернативной версии Manager, может оказаться полезным, когда мы хотим адаптировать класс к ожидаемому интерфейсу, который он не поддерживает, либо отследить или проверить достоверность обращений к методам другого объекта (и действительно мы будем использовать почти идентичный стиль программирования при изучении *декораторов классов* позже в книге).

Кроме того, гипотетический класс Department, подобный показанному ниже, мог бы *агрегировать* другие объекты с целью их трактовки как набора. Временно замените код самотестирования в конце файла person.py, чтобы опробовать прием самостоятельно; это сделано в файле person-department.py, входящем в состав примеров для книги:

```
# Файл person-department.py
# Агрегирование внедренных объектов в составном объекте

class Person:
    ...тот же код, что и ранее...

class Manager(Person):
    ...тот же код, что и ранее...

class Department:
    def __init__(self, *args):
        self.members = list(args)
    def addMember(self, person):
        self.members.append(person)
    def giveRaises(self, percent):
        for person in self.members:
            person.giveRaise(percent)
    def showAll(self):
        for person in self.members:
            print(person)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    tom = Manager('Tom Jones', 50000)
```

```
development = Department(bob, sue)  # Внедрить объекты в составной объект
development.addMember(tom)
development.giveRaises(.10)    # Выполняет giveRaise внедренных объектов
development.showAll()          # Выполняет __repr__ внедренных объектов
```

Во время выполнения метод showAll объекта Department выводит список содержащихся в нем объектов после обновления их состояния в подлинно полиморфной манере с помощью giveRaises:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 110000]
[Person: Tom Jones, 60000]
```

Интересно отметить, что в коде применяются наследование *и* композиция — Department является составным объектом, который внедряет и управляет другими объектами для агрегирования, но сами внедренные объекты Person и Manager для настройки используют наследование. В качестве еще одного примера графический пользовательский интерфейс может похожим образом применять *наследование* для настройки поведения или внешнего вида меток и кнопок, а также *композицию* для построения более крупных пакетов внедряемых виджетов вроде форм ввода, калькуляторов и текстовых редакторов. Используемая структура классов зависит от объектов, которые вы пытаетесь моделировать — по сути, такая возможность моделирования сущностей реального мира считается одной из сильных сторон ООП.

Вопросы проектирования наподобие композиции исследуются в главе 31, так что мы пока отложим дальнейший обзор. Но опять-таки в свете базового механизма ООП на Python наши классы Person и Manager уже отражают всю историю. Однако теперь, когда вы освоили основы ООП, разработка универсальных инструментов для его более легкого применения в своих сценариях часто является естественным следующим шагом — и темой очередного раздела.

Перехват встроенных атрибутов в Python 3.X

Примечание о реализации: в Python 3.X (и в Python 2.X, когда включены классы "нового стиля" Python 3.X) реализованная в главе альтернативная версия класса Manager на основе делегирования (person-composite.py) не сможет перехватывать и делегировать вызовы методов перегрузки операций наподобие __repr__, не переопределяя их. Хотя мы знаем, что __repr__ является единственным таким именем, используемым в нашем специфическом примере, это общая проблема классов, основанных на делегировании.

Вспомните, что встроенные операции вроде вывода и сложения неявно вызывают методы перегрузки операций, такие как __repr__ и __add__. В классах нового стиля Python 3.X встроенные операции подобного рода не маршрутизируют свои неявные выборки атрибутов через обобщенные диспетчеры атрибутов: не вызывается ни метод __getattr__ (выполняется для неопределенных атрибутов), ни родственный ему __getattribute__ (выполняется для всех атрибутов). Именно потому мы вынуждены избыточно переопределять __repr__ в альтернативной версии Manager, чтобы гарантировать направление вывода на внедренный объект Person в Python 3.X.

Закомментируйте данный метод, чтобы увидеть все вживую — экземпляр Manager производит вывод с помощью стандартного инструмента в Python 3.X, но по-прежнему применяет метод __repr__ класса Person в Python 2.X. На самом деле __repr__ из Manager в Python 2.X вообще не требуется, т.к. он реализован для использования нормальных и стандартных (известных как *"классические"*) классов Python 2.X:

```
c:\code> py -3 person-composite.py
[Person: Bob Smith, 0]
...и так далее...
<__main__.Manager object at 0x00000000029AA8D0>

c:\code> py -2 person-composite.py
[Person: Bob Smith, 0]
...и так далее...
[Person: Tom Jones, 60000]
```

Формально так происходит из-за того, что встроенные операции начинают свой неявный поиск имен методов с *экземпляра* в стандартных *классических* классах Python 2.X, но с *класса* в обязательных классах *нового стиля* Python 3.X, полностью пропуская экземпляр. По контрасту с этим явные выборки атрибутов по имени в обеих моделях всегда направляются сначала на экземпляр. В классических классах Python 2.X встроенные атрибуты тоже маршрутизируются подобным образом — например, вывод направляет `__repr__` через `__getattr__`. Вот почему помещение в комментарий метода `__repr__` класса `Manager` не оказывает никакого влияния в Python 2.X: вызов делегируется `Person`. Классы нового стиля также наследуют стандартную реализацию `__repr__` от их автоматического суперкласса `object`, что помешает работе `__getattr__`, но `__getattribute__` нового стиля тоже не перехватывает имя.

Это изменение не является препятствием — классы нового стиля, основанные на делегировании, как правило, могут переопределять методы перегрузки операций для их делегирования вложенным объектам, либо вручную, либо через инструменты или суперклассы. Тем не менее, данная тема слишком сложна, чтобы продолжать ее исследовать в настоящем руководстве, а потому здесь не стоит вдаваться в слишком мелкие детали. В главах 31 и 32 мы еще вернемся к данной теме (в главе 32 классы нового стиля определяются более формально). При обсуждении управления атрибутами в главе 38 и декораторов класса `Private` в главе 39 мы снова продемонстрируем влияние изменения на примеры (и также покажем обходные пути). В почти формальном определении наследования в главе 40 вы увидите, что оно будет фактором особого случая. В языке вроде Python, который поддерживает перехват атрибутов и перегрузку операций, влияние такого изменения может оказаться настолько же обширным, как подразумевает его распространение!

Шаг 6: использование инструментов интроспекции

Давайте внесем одну финальную настройку, прежде чем отправить наши объекты в базу данных. В текущем виде наши классы завершены и демонстрируют большинство основ ООП на Python. Однако все еще остаются две проблемы, которые вероятно должны быть улажены до практического применения классов.

- Во-первых, если вы посмотрите на отображение объектов в той форме, как есть, то заметите, что при выводе экземпляр `tom` класса `Manager` помечается как `Person`. Формально это нельзя считать некорректным, поскольку `Manager` является настроенной и специализированной версией `Person`. Тем не менее, правильнее было бы отображать объект с самым специфическим (т.е. расположенным ниже всех) классом: тем, из которого объект создан.
- Во-вторых, и это более важно, текущий формат отображения показывает только атрибуты, которые мы включили в `__repr__`, что может не учитывать будущие

цели. Например, мы пока не в состоянии проверять, что название должности tom было корректно установлено в mgr конструктором класса Manager, т.к. метод __repr__, реализованный для Person, не выводит данное поле. Хуже того, если мы когда-либо расширим или по-другому изменим набор атрибутов, назначаемых нашим объектам в __init__, нам придется не забыть об обновлении также и метода __repr__ для отображения новых имен, иначе со временем он утратит синхронизацию.

Последний пункт означает, что мы вновь сделали потенциальную добавочную работу для себя в будущем, привнося *избыточность* в код. Поскольку любое рассогласование в __repr__ будет отражаться в выводе программы, такая избыточность может быть более очевидной, чем другие формы, которые мы рассмотрели ранее; однако избегание дополнительной работы в будущем обычно считается *стоящей вещью*.

Специальные атрибуты класса

Мы можем решить обе проблемы с помощью *инструментов интроспекции* Python — специальных атрибутов и функций, которые обеспечивают доступ к внутренностям реализаций объектов. Эти инструменты довольно сложны и, как правило, используются теми, кто создает инструменты для применения другими программистами, а не программистами, которые разрабатывают прикладные приложения. Тем не менее, базовое знание ряда таких инструментов полезно, потому что они позволяют писать код, обрабатывающий классы обобщенным образом. Скажем, в нашем коде есть две привязки, которые могут помочь; они обе были представлены ближе к концу предыдущей главы и использовались в ранних примерах.

- Встроенный атрибут экземпляр.__class__ предоставляет ссылку из экземпляра на класс, из которого экземпляр был создан. В свою очередь классы имеют атрибут __name__, в точности как модули, и последовательность __bases__, обеспечивающую доступ к суперклассам. Мы можем их здесь применять для вывода имени класса, из которого создавался экземпляр, вместо жестко закодированного имени.

- Встроенный атрибут объект.__dict__ предоставляет словарь с одной парой "ключ-значение" для каждого атрибута, присоединенного к объекту пространства имен (включая модули, классы и экземпляры). Поскольку это словарь, мы можем извлекать список его ключей, индексировать по ключу, проходить по его ключам и т.д. для обработки всех атрибутов обобщенным образом. Мы можем использовать такой прием для вывода всех атрибутов в любом экземпляре, а не только тех, которые были жестко закодированы в методах специального отображения, во многом подобно тому, как делалось в инструментах модулей, описанных в главе 25 первого тома.

С первой из указанных категорий мы встречались в главе 25 первого тома, а ниже показан краткий пример применения последних версий классов person.py в интерактивной подсказке Python. Обратите внимание, что мы загружаем Person в интерактивной подсказке посредством оператора from — имена классов находятся в модулях и импортируются из них, в точности как имена функций и другие переменные:

```
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> bob                          # Вызывается __repr__ (не __str__) объекта bob
[Person: Bob Smith, 0]
```

```
>>> print(bob)                    # То же самое: print => __str__ или __repr__
[Person: Bob Smith, 0]

>>> bob.__class__                 # Показывает класс и его имя для bob
<class 'person.Person'>
>>> bob.__class__.__name__
'Person'

>>> list(bob.__dict__.keys()) # Атрибуты на самом деле являются ключами словаря
['pay', 'job', 'name']        # Использовать list для получения списка
                              #   в Python 3.X

>>> for key in bob.__dict__:
        print(key, '=>', bob.__dict__[key])    # Ручная индексация

pay => 0
job => None
name => Bob Smith

>>> for key in bob.__dict__:
        print(key, '=>', getattr(bob, key))    # объект.атрибут, но атрибут -
                                               #   переменная

pay => 0
job => None
name => Bob Smith
```

Как кратко отмечалось в предыдущей главе, некоторые атрибуты, доступные из экземпляра, могут не храниться в словаре `__dict__`, если в классе экземпляра определено средство `__slots__`. Необязательное и относительно неясное средство `__slots__` для классов нового стиля (т.е. для всех классов в Python 3.X) сохраняет атрибуты последовательно в экземпляре, может вообще устранить словарь `__dict__` экземпляра и детально исследуется в главах 31 и 32. Поскольку слоты в действительности принадлежат к классам, а не экземплярам, и в любом случае используются редко, мы можем благополучно проигнорировать их здесь и сосредоточить внимание на нормальном словаре `__dict__`.

Однако имейте в виду, что некоторые программы могут нуждаться в перехвате исключений из-за отсутствующего словаря `__dict__` либо применять `hasattr` для проверки или `getattr` со стандартным значением, если их пользователи развернули слоты. Как будет показано в главе 32, код из следующего раздела потерпит неудачу при использовании классом со слотами (их отсутствия достаточно, чтобы гарантировать наличие `__dict__`), но слоты — и другие "виртуальные" атрибуты — не будут сообщаться как данные экземпляра.

Обобщенный инструмент отображения

Мы можем задействовать эти интерфейсы в суперклассе, который отображает точные имена классов и форматирует все атрибуты экземпляра любого класса. Создайте в своем текстовом редакторе новый файл и поместите в него приведенный далее код — новый независимый модуль по имени `classtools.py`, который реализует такой класс. Из-за того, что его перегруженный метод отображения `__repr__` применяет обобщенные инструменты интроспекции, он будет работать на любом экземпляре независимо от набора атрибутов экземпляра. И поскольку он является классом, то автоматически становится универсальным инструментом форматирования: благодаря наследованию его можно соединять с *любым классом*, в котором желательно использовать такой формат отображения. В качестве дополнительного бонуса, если мы когда-либо захотим из-

менить способ отображения экземпляров, тогда понадобится модифицировать только этот класс, т.к. любой класс, который наследует его метод `__repr__`, при следующем своем запуске подхватит новый формат:

```
# Файл classtools.py (новый)
"Смешанные утилиты и инструменты для классов"

class AttrDisplay:
"""
Предоставляет наследуемый метод перегрузки отображения, который показывает
экземпляры с их именами классов и пары имя=значение для каждого атрибута,
сохраненного в самом экземпляре (но не атрибутов, унаследованных от его классов).
Может соединяться с любым классом и будет работать на любом экземпляре.
"""
def gatherAttrs(self):
    attrs = []
    for key in sorted(self.__dict__):
        attrs.append('%s=%s' % (key, getattr(self, key)))
    return ', '.join(attrs)

def __repr__(self):
    return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())

if __name__ == '__main__':

    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2

    class SubTest(TopTest):
        pass

X, Y = TopTest(), SubTest()    # Создать два экземпляра
print(X)                       # Показать все атрибуты экземпляров
print(Y)                       # Показать имя класса, расположенного ниже всех
```

Обратите внимание на строки документации — т.к. класс является инструментом универсального назначения, мы добавляем функциональную документацию для чтения потенциальными пользователями. Как объяснялось в главе 15 первого тома, строки документации могут размещаться в начале простых функций и модулей, а также классов и любых их методов; функция `help` и инструмент PyDoc извлекают и отображают их автоматически. Мы возвратимся к строкам документации для классов в главе 29.

Когда этот модуль запускается напрямую, его код самотестирования создает два экземпляра и выводит их; определенный здесь метод `__repr__` показывает имя класса экземпляра плюс имена и значения всех его атрибутов, отсортированные по имени атрибута. Вывод одинаков в Python 3.X и 2.X, потому что отображением каждого объекта является одиночная сконструированная строка:

```
C:\code> classtools.py
[TopTest: attr1=0, attr2=1]
[SubTest: attr1=2, attr2=3]
```

Еще одно примечание, касающееся проектирования: поскольку класс применяет `__repr__` вместо `__str__`, его отображение используется во всех контекстах, но клиенты также не будут иметь возможности предоставить альтернативное низкоуровневое отображение — они по-прежнему могут добавлять метод `__str__`, но он при-

меняется только к print и str. В более универсальном инструменте использование взамен __str__ ограничивает границы отображения, но оставляет клиенту возможность добавления __repr__ для вторичного отображения в интерактивной подсказке и во вложенных появлениях. Мы будем следовать этой альтернативной политике при реализации расширенных версий данного класса в главе 31; здесь же мы придерживаемся всеобъемлющего метода __repr__.

Атрибуты экземпляра или атрибуты класса

Если вы достаточно хорошо изучите код самопроверки модуля classtools, то заметите, что его класс отображает только атрибуты экземпляра, присоединенные к объекту self в нижней части дерева наследования; это то, что содержится в словаре __dict__ объекта self. Как и следовало ожидать, мы не видим атрибутов, унаследованных экземпляром от классов, расположенных выше в дереве (например, count в коде самотестирования данного файла – атрибут класса, применяемый в качестве счетчика экземпляров). Унаследованные атрибуты класса присоединяются только к классу, они не копируются в экземпляры.

Если вы когда-нибудь захотите включить также унаследованные атрибуты, тогда можете подняться по ссылке __class__ к классу экземпляра, использовать __dict__ для извлечения атрибутов класса и затем пройти через атрибут __bases__ класса, чтобы подняться к более высоко расположенным суперклассам, при необходимости повторяя процесс. Если вы поклонник простого кода, тогда выполнение встроенного вызова dir на экземпляре вместо применения __dict__ и подъема имело бы во многом такой же эффект, т.к. результаты dir включают унаследованные имена в отсортированном списке результатов. В Python 2.7:

```
>>> from person import Person  # Python 2.X: keys - список, dir показывает меньше
>>> bob = Person('Bob Smith')

>>> bob.__dict__.keys()        # Только атрибуты экземпляра
['pay', 'job', 'name']

>>> dir(bob)                   # Плюс унаследованные атрибуты в классах
['__doc__', '__init__', '__module__', '__repr__', 'giveRaise', 'job',
'lastName', 'name', 'pay']
```

Если вы используете Python 3.X, то вывод будет варьироваться и может оказаться больше, чем ожидалось; вот результаты выполнения последних двух операторов, полученные в Python 3.7 (порядок в списке ключей может меняться от вызова к вызову):

```
>>> list(bob.__dict__.keys())   # Python 3.X: keys - представление, не список
['name', 'job', 'pay']

>>> dir(bob)                   # Python 3.X включает методы типа класса
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
...остальные не показаны: 32 атрибута...
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'gatherAttrs', 'giveRaise', 'job', 'lastName', 'name', 'pay']
```

Код и вывод отличается для Python 2.X и Python 3.X, поскольку dict.keys в Python 3.X не является списком, и dir в Python 3.X возвращает добавочные атрибуты реализации типа класса. Формально dir возвращает в Python 3.X больший объем данных, потому что все классы относятся к "новому стилю" и наследуют крупный набор имен

методов для перегрузки операций от типа класса. Фактически обычно вы захотите отфильтровать большинство имен __X__ из результатов dir в Python 3.X, т.к. они относятся к внутренним деталям реализации, которые в нормальной ситуации отображать нежелательно:

```
>>> len(dir(bob))
32
>>> list(name for name in dir(bob) if not name.startswith('__'))
['gatherAttrs', 'giveRaise', 'job', 'lastName', 'name', 'pay']
```

В интересах пространства мы пока оставим необязательное отображение унаследованных атрибутов класса посредством либо подъема по дереву, либо dir в качестве упражнений для самостоятельной проработки. Дополнительные подсказки по этому вопросу вы можете найти в модуле classtree.py с реализацией инструмента подъема по дереву наследования (глава 29), а также в модуле lister.py с реализациями инструментов для построения списков подъема по дереву (глава 31).

Размышления относительно имен в классах инструментов

И последняя тонкость: поскольку наш класс AttrDisplay в модуле classtools представляет собой универсальный инструмент, предназначенный для смешивания с произвольными классами, необходимо осознавать возможность непреднамеренных *конфликтов имен* с клиентскими классами. В текущем виде предполагается, что клиентские подклассы могут применять его методы __repr__ и gatherAttrs, но последний может выйти за рамки ожидаемого подклассом — если подкласс просто определит собственное имя gatherAttrs, то вероятно нарушит работу нашего класса, т.к. вместо нашей версии будет использоваться версия из подкласса, которая находится ниже в дереве.

Чтобы увидеть это самостоятельно, добавьте gatherAttrs к классу TopTest в коде самотестирования файла. Если только новый метод не идентичен или намеренно не настраивает исходный метод, то наш класс инструмента больше не будет работать так, как планировалось — self.gatherAttrs внутри AttrDisplay находит новый метод из экземпляра TopTest:

```
class TopTest(AttrDisplay):
    ....
    def gatherAttrs(self):    # Замещает метод в AttrDisplay!
        return 'Spam'
```

Это не обязательно плохо — иногда мы хотим, чтобы подклассам были доступны другие методы, либо для прямых вызовов, либо для настройки подобным образом. Тем не менее, если в действительности мы намеревались предоставить только __repr__, то такой подход далек от идеала.

Чтобы свести к минимуму вероятность конфликтов имен подобного рода, программисты на Python часто снабжают имена методов, не предназначенных для внешнего применения, префиксом в виде *одиночного подчеркивания*: _gatherAttrs в нашем случае. Хотя такой прием не обеспечивает полной защиты от неправильного использования (что, если в другом классе тоже определено имя _gatherAttrs?), но его обычно достаточно и он представляет собой распространенное соглашение по именованию для внутренних методов класса.

Лучшим, но менее часто применяемым решением было бы использование *двух подчеркиваний* в начале имени метода: __gatherAttrs. Интерпретатор Python автомати-

чески расширяет имена с двумя подчеркиваниями с целью включения имени содержащего их класса, которое делает имена по-настоящему уникальными при поиске в иерархии наследования. Такое средство обычно называется *псевдозакрытыми атрибутами класса* и более подробно рассматривается в главе 31, где будет приведена расширенная версия данного класса. А пока мы сделаем оба метода доступными.

Финальная форма классов

Теперь для применения построенного обобщенного инструмента в наших классах необходимо лишь импортировать его из модуля, скомбинировать с классом верхнего уровня, используя наследование, и избавиться от специфического метода `__repr__`, который мы реализовали ранее. Новый метод перегрузки отображения будет унаследован всеми экземплярами `Person`, а также `Manager`; класс `Manager` получает `__repr__` от класса `Person`, который приобретает его от класса `AttrDisplay`, реализованного в другом модуле. Ниже показана финальная версия файла `person.py` с внесенными изменениями:

```
# Файл classtools.py (новый)
    ...код был представлен ранее...

# Файл person.py (финальная версия)
"""
Регистрирует и обрабатывает сведения о людях.
Для тестирования классов из этого файла запустите его напрямую.
"""
from classtools import AttrDisplay   # Использовать обобщенный инструмент
                                     #   отображения

class Person(AttrDisplay):           # Комбинирование с классом верхнего уровня
    """
    Создает и обрабатывает записи о людях
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):          # Предполагается, что фамилия указана последней
        return self.name.split()[-1]

    def giveRaise(self, percent):   # Процент должен находиться между 0 и 1
        self.pay = int(self.pay * (1 + percent))

class Manager(Person):
    """
    Настроенная версия Person со специальными требованиями
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)    # Название должности
                                                   #   подразумевается

    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
```

```
print(bob.lastName(), sue.lastName())
sue.giveRaise(.10)
print(sue)
tom = Manager('Tom Jones', 50000)
tom.giveRaise(.10)
print(tom.lastName())
print(tom)
```

Поскольку дополнение является последним, мы добавили несколько *комментариев*, чтобы документировать работу — строки документации для описаний функциональности и комментарии # для небольших примечаний в соответствии с принятыми соглашениями, а также *пустые строки* между методами, чтобы улучшить читабельность кода. В целом такой стиль хорош, когда классы и методы становятся крупными, хотя ранее этого не делалось ради экономии места и уменьшения объема кода для таких небольших классов.

Запустив код прямо сейчас, мы увидим все атрибуты наших объектов, а не только те, которые жестко закодированы в исходном методе `__repr__`. И наша финальная проблема решена: из-за того, что `AttrDisplay` напрямую берет имена классов из экземпляра `self`, каждый объект выводится с именем своего ближайшего (находящегося ниже всех в дереве) класса. Теперь `tom` отображается как объект `Manager`, а не `Person`, и мы можем окончательно удостовериться в корректности заполнения его названия должности в конструкторе `Manager`:

```
C:\code> person.py
[Person: job=None, name=Bob Smith, pay=0]
[Person: job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person: job=dev, name=Sue Jones, pay=110000]
Jones
[Manager: job=mgr, name=Tom Jones, pay=60000]
```

Такое отображение полезнее, чем было ранее. Однако в более широком плане наш класс отображения атрибутов стал *универсальным инструментом*, который посредством наследования мы можем комбинировать с любым классом, чтобы задействовать определяемый им формат отображения. Кроме того, все его клиенты будут автоматически подхватывать будущие изменения, вносимые в наш инструмент. Позже в книге мы встретимся с еще более мощными концепциями инструментов для классов, такими как декораторы и метаклассы; наряду с многочисленными инструментами интроспекции Python они позволяют писать код, который дополняет и управляет классами структурированным и сопровождаемым способом.

Шаг 7 (последний): сохранение объектов в базе данных

К этому моменту наша работа практически завершена. Мы теперь располагаем *двухмодульной системой*, которая не только реализует первоначальные проектные цели по представлению сведений о людях, но и предлагает универсальный инструмент отображения атрибутов, подходящий для применения в других программах. За счет помещения функций и классов в файлы модулей мы гарантируем естественную поддержку ими многократного использования. И за счет реализации программного обеспечения в виде классов мы заставляем их естественным образом поддерживать расширение.

Тем не менее, хотя наши классы работают, как было запланировано, создаваемые ими объекты не являются настоящими записями базы данных. То есть если мы уничтожим сеанс Python, то наши экземпляры исчезнут — они представляют собой временные объекты в памяти и не сохраняются на более постоянном носителе вроде файла, поэтому не будут доступными при будущих запусках программы. Оказывается, сделать объекты экземпляров постоянными несложно с помощью средства Python под названием *постоянство объектов*, которое обеспечивает существование объектов после прекращения работы создавшей их программы. Давайте в качестве финального шага нашего учебного руководства сделаем объект постоянными.

Модули pickle, dbm и shelve

Постоянство объектов реализуется тремя стандартными библиотечными модулями, доступными во всех версиях Python.

pickle

Сериализирует произвольные объекты Python в строку байтов и десериализирует их обратно.

dbm (anydbm в Python 2.X)

Реализует файловую систему с доступом по ключу для хранения строк.

shelve

Применяет предшествующие два модуля для хранения объектов Python в файле по ключу.

Мы упоминали указанные модули в главе 9 при исследовании основ файлов. Они предлагают мощные варианты хранения данных. Хотя мы не можем описать их полностью в учебном руководстве или в книге, они достаточно просты, чтобы для начала работы с ними хватило и краткого введения.

Модуль pickle

Модуль `pickle` является своего рода суперобщим инструментом форматирования и деформатирования объектов: он способен преобразовывать практически произвольный объект Python из памяти в строку байтов, которую позже можно использовать для воссоздания первоначального объекта в памяти. Модуль `pickle` может обрабатывать почти любой создаваемый вами объект — списки, словари, их вложенные комбинации и экземпляры классов. Последние особенно удобны, поскольку они предоставляют данные (атрибуты) и поведение (методы); на самом деле такое сочетание можно считать грубым эквивалентом "записей" и "программ". Из-за такой универсальности модуля `pickle` он может заменить дополнительный код, который пришлось бы иначе писать для создания и разбора специальных представлений объектов внутри текстовых файлов. Сохраняя полученную посредством `pickle` строку объекта, вы фактически делаете его постоянным: просто загрузите и с помощью `pickle` воссоздайте исходный объект.

Модуль shelve

Хотя модуль `pickle` легко применять для сохранения объектов в простых плоских файлах и загрузки из них в более позднее время, модуль `shelve` предлагает добавочный уровень структуры, который позволяет хранить объекты, обработанные `pickle`,

по *ключу*. Модуль shelve транслирует объект в строку посредством pickle и сохраняет полученную строку под ключом в файле dbm; при последующей загрузке shelve извлекает строку по ключу и воссоздает исходный объект в памяти с помощью pickle. Все это кажется запутанным, но для вашего сценария хранилище shelve обработанных посредством pickle объектов выглядит подобно *словарю* — вы индексируете по ключам для извлечения, присваиваете по ключам для сохранения и используете такие словарные инструменты, как len, in и dict.keys для получения информации. Хранилища shelve автоматически отображают словарные операции на объекты, хранящиеся в файле.

В действительности для вашего сценария единственное отличие между хранилищем shelve и нормальным словарем в коде связано с тем, что вы обязаны сначала *открывать* хранилище и *закрывать* его после внесения изменений. Совокупный эффект в том, что хранилище shelve предлагает простую базу данных для сохранения и извлечения собственных объектов Python по ключам, что делает их постоянными между запусками программы. Хранилище shelve не поддерживает инструменты запросов, такие как язык SQL, и оно лишено ряда расширенных возможностей, имеющихся в базах данных производственного уровня (вроде подлинной обработки транзакций). Но собственные объекты Python, сохраненные в хранилище shelve, могут обрабатываться с привлечением всей мощи языка Python после того, как будут извлечены обратно по ключу.

Сохранение объектов в базе данных shelve

Темы модулей pickle и shelve достаточно сложны и мы не собираемся здесь погружаться во все связанные с ними детали; вы можете найти их в руководствах по стандартной библиотеке, а также в книгах, ориентированных на разработку приложений, наподобие *Programming Python* (http://www.oreilly.com/catalog/9780596158101). Однако в коде все выглядит гораздо проще, так что давайте к нему и обратимся.

Мы напишем новый сценарий, который помещает объекты наших классов в хранилище shelve, создав в текстовом редакторе новый файл по имени makedb.py. Наши классы необходимо импортировать в новый файл, чтобы создать несколько экземпляров, подлежащих сохранению. Ранее для загрузки класса в интерактивной подсказке мы применяли оператор from, но на самом деле, как и с функциями и остальными переменными, существуют два способа загрузки класса из файла (имена классов являются переменными, похожими на любые другие, и в этом контексте вообще нет ничего магического):

```
import person                    # Загрузить класс с помощью import
bob = person.Person(...)         # Указывать имя модуля

from person import Person        # Загрузить класс с помощью from
bob = Person(...)                # Использовать имя напрямую
```

Для загрузки класса в сценарии мы будем использовать оператор from, т.к. он требует чуть меньшего объема набора. Ради простоты можно скопировать или набрать заново строки самотестирования из файла person.py, которые создают экземпляры наших классов, чтобы у нас было что сохранять (в рассматриваемом демонстрационном примере не имеет смысла беспокоиться об избыточности тестового кода). Когда у нас есть несколько экземпляров, сохранить их в хранилище shelve практически тривиально. Мы всего лишь импортируем модуль shelve, открываем новое хранилище с внешним именем, присваиваем объекты по ключам в хранилище и по завершении закрываем хранилище, потому что в него были внесены изменения:

```
# Файл makedb.py: сохранение объектов Person в базе данных shelve

from person import Person, Manager      # Загрузить наши классы
bob = Person('Bob Smith')               # Создать объекты для сохранения
sue = Person('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)

import shelve
db = shelve.open('persondb')            # Имя файла, в котором хранятся объекты
for obj in (bob, sue, tom):             # Использовать атрибут name объекта
                                        #   в качестве ключа
    db[obj.name] = obj                  # Сохранить объект в shelve по ключу
db.close()                              # Закрыть после внесения изменений
```

Обратите внимание на то, как мы присваиваем объекты хранилищу `shelve` с применением собственных имен объектов в качестве ключей. Мы поступаем так ради удобства; в хранилище `shelve` *ключом* может быть любая строка, в том числе та, которую мы могли бы создать для обеспечения уникальности, используя такие средства, как идентификаторы процессов и отметки времени (доступные в стандартных библиотечных модулях `os` и `time`). Единственная норма – ключи обязаны быть строками и должны быть уникальными, т.к. мы можем сохранять только один объект на ключ, хотя этот объект может быть списком, словарем или другим объектом, содержащим внутри себя множество объектов.

Фактически значения, сохраняемые под ключами, могут быть объектами Python почти любого вида – встроенными типами вроде строк, списков и словарей, а также экземплярами определяемых пользователем классов, вложенными комбинациями всех подобных типов и т.д. Скажем, атрибуты `name` и `job` наших объектов могли бы быть вложенными словарями и списками, как в предшествующих изданиях настоящей книги (правда, тогда текущий код пришлось бы слегка перепроектировать).

Вот и все, что нужно было сделать – если после запуска сценария никакой вывод не был получен, тогда он, вероятно, выполнил свою работу; мы ничего не выводили, а лишь создавали и сохраняли объекты в базе данных, основанной на файле:

```
C:\code> makedb.py
```

Исследование хранилища shelve в интерактивной подсказке

На этой стадии в текущем каталоге присутствует один или большее количество реальных файлов, имена которых начинаются с `persondb`. Создаваемые в действительности файлы могут варьироваться от платформы к платформе, и точно как во встроенной функции `open`, имя файла в `shelve.open()` считается относительным текущего рабочего каталога, если только оно не включает путь к каталогу. Где бы они ни хранились, эти файлы реализуют файл с доступом по ключу, который содержит представление `pickle` наших трех объектов Python. Не удаляйте упомянутые файлы – они являются вашей базой данных, т.е. именно тем, что вы будете копировать или перемещать при выполнении резервного копирования или переноса хранилища.

Вы можете при желании просмотреть файлы `shelve` в проводнике Windows или в командной оболочке Python, но они представляют собой двоичные файлы хеш-данных, большая часть содержимого которых имеет мало смысла за рамками контекста модуля `shelve`. В случае Python 3.X без установленного дополнительного программного обеспечения наша база данных хранится в трех файлах (в Python 2.X файл всего лишь один, `persondb`, потому что модуль расширения `bsddb` заранее установлен для

shelve; в Python 3.X модуль bsddb является необязательным сторонним дополнением с открытым кодом).

Например, стандартный библиотечный модуль glob в Python позволяет получать в коде перечни файлов в каталогах, чтобы проверять наличие в них файлов, и мы можем открывать файлы в текстовом или двоичном режиме для исследования строк и байтов:

```
>>> import glob
>>> glob.glob('person*')
['person-composite.py', 'person-department.py', 'person.py', 'person.pyc',
'persondb.bak', 'persondb.dat', 'persondb.dir']

>>> print(open('persondb.dir').read())
'Sue Jones', (512, 92)
'Tom Jones', (1024, 91)
'Bob Smith', (0, 80)

>>> print(open('persondb.dat','rb').read())
b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x03\x00\x00\x00jobq\
x03NX\x03\x00
...остальные данные не показаны...
```

Нельзя сказать, что содержимое не поддается расшифровке, но оно может меняться на разных платформах и не совсем подходит для дружественного к пользователям интерфейса базы данных! Чтобы лучше проконтролировать нашу работу, мы можем написать еще один сценарий или заняться исследованием хранилища shelve в интерактивной подсказке. Поскольку хранилища shelve представляют собой объекты Python, содержащие другие объекты Python, мы можем обрабатывать их с помощью нормального синтаксиса Python и режимов разработки. Здесь интерактивная подсказка фактически становится *клиентом базы данных*:

```
>>> import shelve
>>> db = shelve.open('persondb')          # Повторно открыть хранилище shelve

>>> len(db)                               # Сохранены три 'записи'
3
>>> list(db.keys())                       # Ключи являются индексом
['Sue Jones','Tom Jones','Bob Smith']     # list() для создания списка в Python 3.X

>>> bob = db['Bob Smith']                 # Извлечь объект bob по ключу
>>> bob                                   # Выполняет __repr__ из AttrDisplay
[Person: job=None, name=Bob Smith, pay=0]

>>> bob.lastName()                        # Выполняет lastName из Person
'Smith'

>>> for key in db:                        # Проход, извлечение, вывод
        print(key, '=>', db[key])

Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

>>> for key in sorted(db):
        print(key, '=>', db[key])         # Проход по сортированным ключам

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Обратите внимание, что для загрузки либо использования сохраненных объектов импортировать классы Person или Manager необязательно. Скажем, мы можем свободно вызвать метод lastName объекта bob и автоматически получить специальный формат отображения, несмотря на отсутствие в области видимости класса Person, к которому bob относится. Подобное возможно потому, что при сохранении посредством pickle экземпляра класса Python записывает его атрибуты экземпляра self вместе с именем класса, из которого он был создан, и модуля, где класс находится. Когда объект bob позже извлекается из хранилища shelve и воссоздается с помощью pickle, то Python автоматически повторно импортирует класс и связывает с ним bob.

Результатом такой схемы оказывается то, что экземпляры класса автоматически обзаводятся поведением их класса при загрузке в будущем. Мы должны импортировать классы, только чтобы создать новые экземпляры, но не обрабатывать существующие. Хотя так сделано умышленно, схема влечет за собой смешанные последствия.

- Недостаток в том, что классы и файлы их модулей обязаны быть импортируемыми, когда экземпляр позже загружается. Более формально классы, допускающие обработку посредством pickle, должны записываться на верхнем уровне файла модуля, который доступен в каталоге из пути поиска модулей sys.path (и не находиться в модуле __main__ самого верхнего файла сценария, если только при использовании они не всегда присутствуют в данном модуле). Из-за такого требования к внешнему файлу модуля в некоторых приложениях принимается решение обрабатывать с помощью pickle более простые объекты, подобные словарям или спискам, особенно если они передаются по сети.
- Преимущество в том, что изменения в файле исходного кода класса автоматически подхватываются, когда экземпляры этого класса загружаются снова; часто нет необходимости обновлять сами сохраненные объекты, т.к. обновление кода их класса изменяет поведение.

Хранилища shelve также обладают хорошо известными ограничениями (некоторые из них упоминаются в соображениях относительно баз данных в конце главы). Тем не менее, для простого хранилища объектов модули shelve и pickle являются удивительно легкими для применения инструментами.

Обновление объектов в хранилище shelve

Теперь о последнем сценарии: давайте напишем программу, которая при каждом запуске обновляет экземпляр (запись), чтобы доказать *постоянство* наших объектов — что их текущие значения доступны в любой момент, когда программа на Python выполняется. Файл updatedb.py, код которого приведен ниже, выводит содержимое базы данных и каждый раз предоставляет повышение одному из сохраненных объектов. Если вы отследите происходящее, то заметите, что мы получаем много возможностей "бесплатно" — вывод наших объектов автоматически задействует универсальный метод перегрузки __repr__, и мы предоставляем повышения, вызывая реализованный ранее метод giveRaise. Все они "просто работают" для объектов, основанных на модели наследования ООП, даже когда объекты располагаются в файле:

```
# Файл updatedb.py: обновление объекта Person в базе данных

import shelve
db = shelve.open('persondb')        # Заново открыть хранилище shelve
                                    #   с тем же самым именем файла
```

```
for key in sorted(db):            # Проход для отображения объектов из базы данных
  print(key, '\t=>', db[key]) # Выводит в специальном формате

sue = db['Sue Jones']             # Индексация по ключу с целью извлечения
sue.giveRaise(.10)                # Обновление в памяти, используя метод класса
db['Sue Jones'] = sue             # Присваивание по ключу для обновления
                                  #   в хранилище shelve
db.close()                        # Закрытие после внесения обновлений
```

Поскольку сценарий выводит содержимое базы данных при начальном запуске, чтобы увидеть, изменился ли объект, мы должны запустить его, по меньшей мере, дважды. Далее сценарий показан в действии, отображая все записи и увеличивая заработную плату объекта sue каждый раз, когда он запускается (довольно хороший сценарий для sue... возможно, его стоило бы запланировать к запуску на регулярной основе как задание cron?):

```
C:\code> updatedb.py
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

C:\code> updatedb.py
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

C:\code> updatedb.py
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=121000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

C:\code> updatedb.py
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=133100]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Опять-таки то, что мы видим, является результатом функционирования инструментов shelve и pickle, которые мы получаем от Python, и поведения, реализованного в наших классах. Мы снова можем проверить работу нашего сценария в интерактивной подсказке — эквиваленте клиента базы данных shelve:

```
C:\code> python
>>> import shelve
>>> db = shelve.open('persondb')    # Повторно открыть базу данных
>>> rec = db['Sue Jones']           # Извлечь объект по ключу
>>> rec
[Person: job=dev, name=Sue Jones, pay=146410]
>>> rec.lastName()
'Jones'
>>> rec.pay
146410
```

Еще один пример постоянства объектов описан во врезке "Что потребует внимания: классы и постоянство" в главе 31. Он сохраняет несколько больший составной объект в плоском файле с помощью pickle вместо shelve, но эффект аналогичен. Дополнительные сведения, касающиеся работы с модулями pickle и shelve, ищите в главах 9 (основы файлов) и 37 (изменения строковых инструментов Python 3.X), в других книгах, а также в руководствах по Python.

Указания на будущее

На этом наше учебное руководство завершается. К настоящему моменту вы видели в действии все основы механизма ООП в Python и изучили способы избегания избыточности и связанных с нею проблем сопровождения кода. Были построены полнофункциональные классы, выполняющие реальную работу. В качестве дополнительного бонуса они были сделаны настоящими записями базы данных за счет помещения в хранилище `shelve`, так что информация в них сохранялась на постоянной основе.

Разумеется, мы могли бы здесь исследовать и многое другое, например, расширить классы, сделав их более реалистичными, добавить к ним новые линии поведения и т.д. Скажем, на практике операция предоставления повышения должна проверять, находится ли процент повышения между 0 и 1 – расширение, которое мы добавим при обсуждении декораторов позже в книге. Рассмотренный пример можно было бы превратить в базу данных личных контактов за счет изменения сведений, хранящихся в объектах, а также методов классов, используемых для их обработки. Мы оставляем это предполагаемое упражнение полностью открытым для вашего воображения.

Мы могли бы также расширить масштаб охвата и применять инструменты, которые либо поступают вместе с Python, либо свободно доступны в мире открытого кода.

Графические пользовательские интерфейсы

В том виде, как есть, мы можем обрабатывать базу данных только с помощью основанного на командах интерфейса интерактивной подсказки и сценариев. Мы могли бы потрудиться над расширением удобства эксплуатации нашей базы данных объектов, добавив настольный графический пользовательский интерфейс для просмотра и обновления ее записей. Графические пользовательские интерфейсы можно строить переносимым образом посредством либо стандартной библиотечной поддержки `tkinter` (`Tkinter` в Python 2.X), либо сторонних инструментальных комплектов, таких как wxPython и PyQt. Комплект `tkinter` поставляется вместе с Python, позволяет быстро строить простые графические пользовательские интерфейсы, и он идеален для изучения методик программирования интерфейсов подобного рода; wxPython и PyQt сложнее в использовании, но часто в итоге дают интерфейсы более высокого качества.

Веб-сайты

Хотя графические пользовательские интерфейсы являются удобными и быстрыми, веб-сеть трудно превзойти с точки зрения доступности. Для просмотра и обновления записей мы могли бы также реализовать веб-сайт взамен или в дополнение к графическому пользовательскому интерфейсу и интерактивной подсказке. Веб-сайты можно создавать с помощью либо базовых инструментов для написания сценариев CGI, имеющихся в составе Python, либо полномасштабных сторонних веб-фреймворков вроде Django, TurboGears, Pylons, web2Py, Zope или Google App Engine. В веб-сети ваши данные по-прежнему могут находиться в хранилище `shelve`, в файле `pickle` или в другой среде, основанной на Python. Сценарии обработки данных запускаются автоматически на сервере в ответ на запросы из веб-браузеров и других клиентов, и они производят HTML-разметку для взаимодействия с пользователем, либо напрямую, либо через API-интерфейсы фреймворка. Системы обогащенных Интернет-приложений (Rich Internet application – RIA), такие как Silverlight и pyjamas, также пытаются объединить взаимодействие, подобное обеспечиваемому графическими пользовательскими интерфейсами, с развертыванием в веб-сети.

Веб-службы

Хотя веб-клиенты часто способны проводить разбор информации в ответах от веб-сайтов (методика, красочно называемая "анализом экранных данных"), мы могли бы двинуться дальше и обеспечить более прямой способ извлечения записей через веб-сеть посредством интерфейса веб-служб, такого как SOAP или вызовы XML-RPC — API-интерфейсов, поддерживаемых либо самим Python, либо сторонними инструментами с открытым кодом, которые в целом отображают данные в формат XML и обратно с целью передачи. Для сценариев на Python подобного рода API-интерфейсы возвращают данные более непосредственно, чем текст, внедренный в HTML-разметку страницы ответа.

Базы данных

Если наша база данных становится объемной или важной, тогда со временем мы могли бы перейти от модуля `shelve` к более полнофункциональному механизму хранения. Им может быть система управления объектно-ориентированными базами данных с открытым кодом ZODB или более традиционная система управления реляционными базами данных на основе SQL, такая как MySQL, Oracle или PostgreSQL. Сам Python поставляется со встроенным модулем внутрипроцессной системы управления базами данных SQLite, но в веб-сети свободно доступны другие варианты с открытым кодом. Например, система ZODB похожа на модуль `shelve` из Python, но лишена множества его ограничений, лучше поддерживает крупные базы данных, параллельные обновления, обработку транзакций и автоматическую сквозную запись при изменениях в памяти (хранилища `shelve` могут кешировать объекты и сбрасывать их на диск во время закрытия посредством параметра `writeback`, но с этим связаны ограничения). Системы на основе SQL, подобные MySQL, предлагают инструменты производственного уровня для хранилища в виде базы данных и могут напрямую применяться в сценарии на Python. Как известно из главы 9 первого тома, MongoDB обеспечивает альтернативный подход, предусматривающий хранение документов JSON, которые близко напоминают словари и списки Python, но в отличие от данных `pickle` нейтральны к языку.

Средства объектно-реляционного отображения

В случае перевода хранилища на систему правления реляционными базами данных мы не должны приносить в жертву инструменты ООП в Python. Средства объектно-реляционного отображения (object-relational mapper — ORM) вроде SQLObject и SQLAlchemy способны автоматически отображать реляционные таблицы и строки на классы и экземпляры Python и обратно, так что мы можем обрабатывать сохраненные данные с использованием нормального синтаксиса классов Python. Такой подход предлагает альтернативу системам управления объектно-ориентированными базами данных наподобие `shelve` и ZODB и задействует мощь реляционных баз данных и модели классов Python.

Хотя я надеюсь, что предложенное введение разожгло у вас интерес к дальнейшим исследованиям, все эти темы, конечно же, выходят далеко за рамки данного руководства и книги в целом. Если вы хотите продолжить изучение самостоятельно, тогда ищите информацию в веб-сети, в руководствах по стандартной библиотеке Python и в ориентированных на разработку приложений книгах, таких как *Programming Python* (`http://oreilly.com/catalog/9780596009250/`). В последней начатый в главе при-

мер продолжается демонстрацией того, как добавить графический пользовательский интерфейс и веб-сайт поверх базы данных, чтобы сделать возможным удобный просмотр и обновление записей экземпляров. А теперь давайте возвратимся к основам классов и закончим остаток истории с языком Python.

Резюме

В главе мы исследовали все основы классов Python и ООП в действии, построив простой, но реалистичный пример шаг за шагом. Мы добавили конструкторы, методы, перегрузку операций, настройку с помощью подклассов и инструменты интроспекции, а также попутно ознакомились с другими концепциями, такими как композиция, делегирование и полиморфизм.

В итоге мы взяли объекты, созданные нашими классами, и сделали их постоянными за счет сохранения в базе данных объектов `shelve` — легкой в применении системы для сохранения и извлечения собственных объектов Python по ключу. Во время изучения основ классов мы также ознакомились с несколькими способами вынесения кода, призванными сократить избыточность и минимизировать будущие затраты на сопровождение. Наконец, был приведен краткий обзор подходов к расширению кода с помощью инструментов для реализации приложений, в числе которых графические пользовательские интерфейсы и базы данных, раскрываемые в отдельных книгах.

В последующих главах этой части книги мы возвратимся к изучению деталей, лежащих в основе модели классов Python, и исследованию ее применимости к ряду проектных концепций, используемых для объединения классов в более крупных программах. Однако прежде чем двигаться дальше, ответьте на контрольные вопросы главы. Поскольку в главе уже было проделано много практической работы, мы завершаем главу набором вопросов в основном по теории, которые заставят вас пройтись по коду и обдумать стоящие за ним идеи.

Проверьте свои знания: контрольные вопросы

1. Когда мы извлекаем объект `Manager` из хранилища `shelve` и выводим его, то откуда поступает логика форматирования отображения?
2. Когда мы извлекаем объект `Person` из хранилища `shelve`, не импортируя его модуль, то каким образом объект узнает, что он имеет метод `giveRaise`, который мы можем вызвать?
3. Почему настолько важно перемещать обработку внутрь методов вместо ее жесткого кодирования за пределами класса?
4. Почему лучше настраивать за счет создания подклассов, а не копировать исходный код и модифицировать копию?
5. Почему для выполнения стандартных действий лучше вызывать метод суперкласса, а не копировать и модифицировать их код в подклассе?
6. Почему лучше применять инструменты вроде `__dict__`, которые позволяют обрабатывать объекты обобщенно, чем писать дополнительный специализированный код для каждой разновидности класса?
7. Опишите в общих чертах, когда вы могли бы выбрать использование внедрения и композиции объектов вместо наследования?

8. Что бы вы изменили, если бы объекты, реализованные в настоящей главе, применяли словарь для имен и список для названий должностей, как в похожих примерах, приводимых ранее в книге?
9. Как бы вы модифицировали классы в этой главе для реализации базы данных личных контактов в Python?

Проверьте свои знания: ответы

1. В финальной версии наших классов `Manager` в конечном итоге наследует свой метод вывода `__repr__` от класса `AttrDisplay` из отдельного модуля `classtools`, находящегося двумя уровнями выше в дереве классов. Сам класс `Manager` не имеет такого метода, поэтому поиск в иерархии наследования поднимается до его суперкласса `Person`; из-за того, что там тоже нет метода `__repr__`, поиск поднимается выше и обнаруживает его в классе `AttrDisplay`. Имена классов, указанные внутри круглых скобок в строке заголовка оператора `class`, предоставляют ссылки на более высоко расположенные суперклассы.
2. Хранилища `shelve` (в действительности используемый ими модуль `pickle`) автоматически повторно связывают экземпляр с классом, из которого он был создан, когда экземпляр позже загружается обратно в память. Python внутренне заново импортирует класс из его модуля, создает экземпляр с сохраненными атрибутами и устанавливает ссылку `__class__` экземпляра, чтобы она указывала на первоначальный класс. Таким образом, загруженные экземпляры автоматически получают все свои первоначальные методы (наподобие `lastName`, `giveRaise` и `__repr__`), даже притом, что мы не импортировали класс экземпляра в текущую область видимости.
3. Важно переносить обработку внутрь методов, чтобы в будущем приходилось изменять только одну копию и методы могли выполняться на любом экземпляре. Это понятие *инкапсуляции* в Python — помещение логики в оболочку интерфейсов для лучшей поддержки будущего сопровождения кода. Если вы не поступите так, то создадите избыточность кода, которая может увеличить трудозатраты по мере развития кода в будущем.
4. Настройка с помощью создания подклассов сокращает объем усилий, требующихся для разработки. При ООП мы пишем код, *настраивая* то, что уже было сделано, вместо копирования либо изменения существующего кода. На самом деле это "основной смысл" ООП — из-за возможности легкого расширения предыдущей работы путем реализации новых подклассов мы можем задействовать то, что было сделано ранее. Поступать так гораздо лучше, чем либо начинать каждый раз с нуля, либо вводить многочисленные избыточные копии кода, которые все могут потребовать обновления в будущем.
5. Копирование и модификация кода *удваивает* потенциальные трудозатраты в будущем независимо от контекста. Если подкласс нуждается в выполнении стандартных действий, реализованных в методе суперкласса, то намного лучше обратиться к исходному методу через имя суперкласса, чем копировать его код. Это также остается справедливым для конструкторов суперкласса. Опять-таки копирование кода создает избыточность, которая с развитием кода превращается в крупную проблему.

6. Обобщенные инструменты в состоянии избежать жестко закодированных решений, которые должны быть синхронизированы с остальной частью класса по мере того, как класс развивается с течением времени. Например, обобщенный метод вывода `__repr__` не придется обновлять каждый раз, когда к экземплярам добавляется новый атрибут в конструкторе `__init__`. Вдобавок обобщенный метод `print`, наследуемый всеми классами, обнаруживается и должен быть модифицирован только в одном месте — изменения в обобщенной версии подхватываются всеми классами, которые унаследованы от обобщенного класса. И снова устранение *избыточности* кода сокращает будущие усилия по разработке; это одно из основных полезных качеств, привносимых классами.

7. Наследование лучше всего подходит при реализации расширений, основанных на прямой настройке (подобных нашей специализации `Manager` из класса `Person`). Композиция хорошо подходит в сценариях, где множество объектов агрегируются в единое целое и управляются классом уровня контроллера. Наследование передает вызовы *вверх* для многократного использования, а композиция передает их *вниз* для делегирования. Наследование и композиция не являются взаимно исключающими; зачастую объекты, внедренные в контроллер, сами представляют собой настройки, основанные на наследовании.

8. Не особенно много, т.к. это был действительно первый прототип, но метод `lastName` потребовалось бы обновить для нового формата имени; в конструкторе `Person` пришлось бы изменить стандартное значение названия должности на пустой список; и в конструкторе класса `Manager` вероятно понадобилось бы передавать список названий должностей вместо одиночной строки (разумеется, нужно также изменить код самотестирования). Хорошая новость в том, что упомянутые изменения необходимо вносить всего лишь в одном месте — в наших классах, где инкапсулированы такие детали. Сценарий базы данных должен работать в том виде, как есть, поскольку хранилища `shelve` поддерживают произвольно глубоко вложенные данные.

9. Классы в настоящей главе можно было бы применять в качестве стереотипного "шаблонного" кода для реализации различных типов баз данных. По существу вы можете изменить их назначение, модифицируя конструкторы для регистрации других атрибутов и предоставляя любые методы, которые подходят целевому приложению. Скажем, для базы данных контактов вы могли бы использовать такие атрибуты, как `name`, `address`, `birthday`, `phone`, `email` и т.д., и подходящие для этой цели методы. Например, метод по имени `sendmail` мог бы применять стандартный библиотечный модуль `smtplib` в Python для отправки сообщения электронной почты одному из контактов автоматически при вызове (дополнительные сведения об инструментах подобного рода ищите в руководствах по Python или в книгах, посвященных разработке приложений). Написанный здесь инструмент `AttrDisplay` можно было бы дословно использовать для вывода объектов, поскольку он намеренно сделан обобщенным. Большую часть кода базы данных `shelve` можно было бы применять для хранения ваших объектов, внеся лишь незначительные изменения.

ГЛАВА 29

Детали реализации классов

Если вы еще не полностью уловили суть ООП в Python, то не переживайте; после первого тура мы собираемся погрузиться чуть глубже и заняться исследованием представленных ранее концепций. В этой и следующей главах мы еще раз взглянем на механизм классов. Здесь мы займемся изучением классов, методов и наследования, формализуя и расширяя ряд идей реализации, которые были введены в главе 27. Поскольку класс является нашим последним инструментом пространств имен, мы также подведем итоги по концепциям пространств имен и областей видимости Python.

В следующей главе продолжается углубленный проход по механизму классов за счет раскрытия одного специфического аспекта: перегрузки операций. Помимо представления дополнительных деталей в текущей и следующей главах также предоставляется возможность исследования ряда более крупных классов, чем те, которые были исследованы до сих пор.

Замечание по содержанию: если вы читаете последовательно, тогда некоторые материалы главы окажутся обзором и сводкой по темам, представленным в учебном примере из предыдущей главы, которые здесь пересматриваются с точки зрения языка, с меньшими и более автономными примерами для читателей, не знакомых с ООП. У вас может возникнуть искушение пропустить часть данной главы, но обязательно просмотрите обзор пространств имен, т.к. в нем объясняются некоторые тонкости в модели классов Python.

Оператор `class`

Хотя оператор `class` в Python на первый взгляд может выглядеть похожим на инструменты в других языках ООП, при ближайшем рассмотрении выясняется, что он совершенно отличается о того, к чему привыкли программисты. Например, как и в C++, оператор `class` представляет собой главный инструмент ООП в Python, но в отличие от C++ оператор `class` языка Python `class` не является объявлением. Подобно `def` оператор `class` создает объекты и неявно присваивает их — при выполнении он генерирует объект класса и сохраняет ссылку на него в имени, используемом в заголовке. Также подобно `def` оператор `class` относится к настоящему исполняемому коду — ваш класс не существует до тех пор, пока Python не достигнет и не выполнит оператор `class`, который его определяет. Обычно это происходит во время импортирования модуля, содержащего оператор `class`, но не ранее.

Общая форма

Оператор class является составным, с телом, состоящим из операторов, которые обычно размещаются с отступом под заголовком. В заголовке внутри круглых скобок после имени класса приводится список суперклассов, разделенных запятыми. Указание более одного суперкласса приводит к множественному наследованию, которое мы формально обсудим в главе 31. Ниже показана общая форма оператора:

```
class имя(суперкласс,...):          # Присваивание имени
    атрибут = значение              # Разделяемые данные класса
    def метод(self,...):            # Методы
        self.атрибут = значение     # Данные экземпляра
```

Любые присваивания внутри оператора class генерируют атрибуты класса, а особым образом именованные методы перегружают операции; скажем, функция по имени __init__ вызывается на стадии конструирования объекта экземпляра, если она определена.

Пример

Как мы уже видели, классы главным образом представляют собой всего лишь *пространства имен* — т.е. инструменты для определения имен (атрибутов), которые экспортируют данные и логику клиентам. Оператор class фактически определяет пространство имен. Как и в файле модуля, вложенные в тело class операторы создают атрибуты класса. Когда Python выполняет оператор class (не вызов класса), он запускает все операторы в теле от начала до конца. Присваивания, происходящие во время такого процесса, создают имена в локальной области видимости класса, которые становятся атрибутами в ассоциированном объекте класса. По этой причине классы напоминают и *модули*, и *функции*:

- подобно функциям операторы class являются локальными областями видимости, где находятся имена, созданные вложенными присваиваниями;
- подобно именам в модуле имена, присвоенные в операторе class, становятся атрибутами в объекте класса.

Главное отличие классов связано с тем, что их пространства имен формируют также основу *наследования* в Python; указанные атрибуты, которые не обнаруживаются в объекте класса или экземпляра, извлекаются из других классов.

Поскольку class представляет собой составной оператор, в его тело могут вкладываться операторы любого вида, т.е. print, присваивания, if, def и т.д. Все операторы внутри class выполняются при выполнении самого оператора class (не тогда, когда позже происходит вызов класса для создания экземпляра). Обычно операторы присваивания внутри оператора class создают атрибуты данных, а вложенные операторы def — атрибуты методов. Однако в общем случае любой вид присваивания значения имени на верхнем уровне оператора class создает атрибут с таким именем в результирующем объекте класса.

Например, присваивания значений простым объектам, отличным от функций, производят *атрибуты данных*, разделяемые всеми экземплярами:

```
>>> class SharedData:
        spam = 42                   # Генерирует атрибут данных класса

>>> x = SharedData()                # Создание двух экземпляров
>>> y = SharedData()
```

```
>>> x.spam, y.spam      # Они наследуют и разделяют spam
                        #  (известный как SharedData.spam)
(42, 42)
```

Поскольку имени spam присваивается значение на верхнем уровне оператора class, оно присоединяется к классу, а потому будет разделяться всеми экземплярами. Мы можем изменять его через имя класса и ссылаться посредством либо экземпляров, либо самого класса[1]:

```
>>> SharedData.spam = 99
>>> x.spam, y.spam, SharedData.spam
(99, 99, 99)
```

Такие атрибуты классов могут применяться для управления информацией, которая охватывает все экземпляры — скажем, счетчиком количества сгенерированных экземпляров (мы разовьем эту идею с помощью примера в главе 32). А пока посмотрим, что происходит в случае присваивания значения имени spam через экземпляр вместо класса:

```
>>> x.spam = 88
>>> x.spam, y.spam, SharedData.spam
(88, 99, 99)
```

Присваивания значений атрибутам экземпляра создают либо изменяют имена в экземпляре, а не в разделяемом классе. В более общем смысле поиск в иерархии наследования инициируется только при *ссылке* на атрибуты, не в случае присваивания: присваивание значения атрибуту объекта всегда изменяет данный объект, но никакой другой[2]. Например, y.spam ищется в классе через наследование, но присваивание x.spam присоединяет имя к самому объекту x.

Ниже приведен более полный пример такого поведения, который сохраняет то же самое имя в двух местах. Пусть мы запускаем следующий класс:

```
class MixedNames:                        # Определить класс
    data = 'spam'                        # Присвоить значение атрибуту класса
    def __init__(self, value):           # Присвоить значение имени метода
        self.data = value                # Присвоить значение атрибуту экземпляра
    def display(self):
        print(self.data, MixedNames.data)  # Атрибут экземпляра, атрибут класса
```

Класс MixedNames содержит два оператора def, которые привязывают атрибуты класса к функциям методов.

---

[1] Если вы использовали язык C++, тогда можете счесть это похожим на понятие "статических" данных-членов C++ — членов, которые хранятся в классе независимо от экземпляров. В Python с ними не связано ничего особенного: все атрибуты класса представляют собой просто имена, присвоенные в операторе class, будь они ссылками на функции ("методы" C++) или на что-то другое ("члены" C++). В главе 32 мы также встретим статические методы Python (родственные таковым в C++), которые являются всего лишь функциями, обычно обрабатывающими атрибуты классов.

[2] Если только класс с помощью метода перегрузки операции __setattr__ (обсуждается в главе 30) не переопределяет операцию присваивания атрибутам, чтобы она делала что-то уникальное, или не использует расширенные инструменты для работы с атрибутами, такие как *свойства* и *дескрипторы* (рассматриваются в главах 32 и 38). В большей части этой главы представлен нормальный случай, которого на данной стадии достаточно, но как будет показано позже, привязки Python позволяют программам часто отклоняться от нормы.

Он также содержит оператор присваивания =; поскольку этот оператор присваивает значение имени data внутри class, оно существует в локальной области видимости класса и становится атрибутом объекта класса. Как и все атрибуты класса, data наследуется и разделяется всеми экземплярами класса, которые не имеют собственных атрибутов data.

Когда мы создаем экземпляры класса MixedNames, имя data присоединяется к ним путем присваивания self.data в методе конструктора:

```
>>> x = MixedNames(1)            # Создать два объекта экземпляра
>>> y = MixedNames(2)            # Каждый имеет собственный атрибут data
>>> x.display(); y.display()     # Атрибуты self.data отличаются,
                                 #   MixedNames.data - тот же самый
1 spam
2 spam
```

Совокупный результат в том, что атрибут data присутствует в двух местах: в объектах экземпляров (создан присваиванием self.data в __init__) и в классе, от которого они наследуют имена (создан присваиванием data в class). Метод display класса выводит обе версии, сначала версию, уточненную посредством экземпляра self, и затем версию класса.

За счет применения таких методик для сохранения атрибутов в разных объектах мы определяем их область видимости. Когда имена присоединены к классам, они разделяются; в экземплярах имена фиксируют данные для каждого экземпляра, не разделяя поведение или данные. Несмотря на то что поиск в иерархии наследования находит для нас имена, мы всегда можем извлечь атрибут откуда угодно из дерева, получая доступ к желаемому объекту напрямую.

В предыдущем примере указание x.data или self.data возвратит имя экземпляра, которое в нормальной ситуации скрывает такое же имя в классе; тем не менее, MixedNames.data явно захватывает версию имени класса. В следующем разделе описана одна из наиболее распространенных ролей для таких кодовых схем, а также объясняется, как мы ее развернули в предыдущей главе.

Методы

Поскольку вы уже осведомлены о функциях, вам также известно о методах в классах. Методы являются просто объектами функций, которые создаются операторами def, вложенными в тело оператора class. С абстрактной точки зрения методы предоставляют поведение для наследования объектами экземпляров. С точки зрения программирования методы работают в точности таким же образом, как и простые функции, но с одним критически важным исключением: первый аргумент метода всегда получает объект экземпляра, который представляет собой подразумеваемый объект вызова метода. Другими словами, Python автоматически отображает вызовы методов экземпляра на функции методов класса, как показано далее. Вызовы методов, выполненные через экземпляр:

```
экземпляр.метод(аргументы...)
```

автоматически транслируются в вызовы функций методов класса в следующей форме:

```
класс.метод(экземпляр, аргументы...)
```

где Python определяет класс, находя имя метода с использованием процедуры поиска в иерархии наследования. На самом деле в Python допустимы обе формы вызова.

Помимо нормального наследования имен атрибутов методов специальный первый аргумент является единственной реальной магией, стоящей за вызовами методов. В методе класса первый аргумент по соглашению называется self (формально важна только его позиция, но не имя). Аргумент self снабжает метод привязкой к экземпляру, на котором производился вызов — из-за того, что классы генерируют много объектов экземпляров, им необходимо применять этот аргумент для управления данными, которые варьируются от экземпляра к экземпляру.

Программисты на C++ могут счесть аргумент self в Python похожим на указатель this в C++. Однако в Python аргумент self всегда будет явным в коде: методы всегда обязаны проходить через self, чтобы извлечь или изменить атрибуты экземпляра, обрабатываемого текущим вызовом метода. Явная природа self умышленна — присутствие этого имени делает очевидным то, что вы используете в своем сценарии имена атрибутов экземпляра, а не имена в локальной или глобальной области видимости.

Пример метода

Чтобы пролить свет на упомянутые концепции, давайте перейдем к рассмотрению примера. Пусть мы определили такой класс:

```
class NextClass:                         # Определить класс
    def printer(self, text):             # Определить метод
        self.message = text              # Изменить экземпляр
        print(self.message)              # Получить доступ к экземпляру
```

Имя printer ссылается на объект функции; поскольку оно присваивается в области видимости оператора class, то становится атрибутом объекта класса и наследуется каждым экземпляром, созданным из класса. Обычно из-за того, что методы вроде printer предназначены для обработки экземпляров, мы вызываем их через экземпляры:

```
>>> x = NextClass()                      # Создать экземпляр
>>> x.printer('instance call')           # Вызвать его метод
instance call
>>> x.message                            # Экземпляр изменился
'instance call'
```

Когда мы вызываем метод, подобным образом уточняя экземпляр, метод printer сначала ищется в иерархии наследования и затем его аргументу self автоматически присваивается объект экземпляра (x); аргумент text получает строку, переданную при вызове ('instance call'). Обратите внимание, что поскольку Python автоматически передает первый аргумент self, то нам фактически понадобится передать один аргумент. Внутри printer имя self применяется для доступа или установки данных экземпляра, т.к. оно ссылается на экземпляр, который в текущий момент обрабатывается.

Тем не менее, как будет показано, методы могут вызываться одним из двух способов — через экземпляр или через сам класс. Например, мы также можем вызывать printer, указывая имя класса, при условии явной передачи экземпляра в аргументе self:

```
>>> NextClass.printer(x, 'class call') # Прямой вызов через класс
class call
>>> x.message                          # Экземпляр снова изменяется
'class call'
```

Вызовы, маршрутизируемые через экземпляр и класс, дают в точности тот же самый эффект до тех пор, пока мы передаем в форме вызова с классом один и тот же экземпляр. На самом деле вы по умолчанию получите сообщение об ошибке, если попытаетесь вызывать метод без экземпляра:

```
>>> NextClass.printer('bad call')
TypeError: unbound method printer() must be called with NextClass instance...
Ошибка типа: несвязанный метод printer() должен вызываться с экземпляром NextClass...
```

Вызов конструкторов суперклассов

Методы обычно вызываются через экземпляры. Однако вызовы методов через класс обнаруживаются в разнообразных специальных ролях. Один распространенный сценарий касается метода конструктора. Подобно всем атрибутам метод `__init__` ищется в иерархии наследования. Другими словами, на стадии конструирования Python ищет и вызывает только *один* метод `__init__`. Если конструкторам подклассов нужна гарантия того, что также выполняется логика стадии конструирования суперкласса, тогда в общем случае они обязаны явно вызывать метод `__init__` суперкласса через класс:

```
class Super:
    def __init__(self, x):
        ...стандартный код...

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)   # Выполнить метод __init__ суперкласса
        ...специальный код...     # Выполнить специальные действия инициализации

I = Sub(1, 2)
```

Это один из нескольких контекстов, в которых ваш код, вероятно, вызовет метод перегрузки операции напрямую. Естественно, вы должны вызывать конструктор суперкласса подобным образом, только если действительно *хотите*, чтобы он выполнился – без такого вызова подкласс заместит его полностью. Более реалистичная иллюстрация данной методики приводилась в форме примера класса Manager в предыдущей главе[3].

Другие возможности вызова методов

Описанная схема вызова методов через класс представляет собой общую основу расширения – вместо полной замены – поведения унаследованных методов. Она требует передачи явного экземпляра, т.к. по умолчанию это делают все методы. Формально причина в том, что в отсутствие любого специального кода методы являются *методами экземпляров*.

В главе 32 вы также ознакомитесь с более новым вариантом, появившимся в версии Python 2.2 – *статическими методами*, которые делают возможным написание методов, не ожидающих получения в своих первых аргументах объектов экземпляров. Такие методы способны действовать подобно простым функциям, не связанным с экземплярами, с именами, считающимися локальными по отношению к классам, внутри которых они реализованы, и могут применяться для управления данными классов. Связанная

[3] В качестве связанного замечания: внутри одного класса вы также можете реализовать множество методов `__init__`, но будет использоваться только последнее определение; за дополнительными сведениями по нескольким определениям методов обращайтесь в главу 31.

концепция, рассматриваемая в той же главе 32, *метод класса*, предусматривает передачу класса взамен экземпляра во время вызова метода; она может использоваться для управления данными каждого класса и ее наличие подразумевается в метаклассах.

Тем не менее, указанные расширения являются более сложными и обычно необязательными. В нормальной ситуации методу всегда должен передаваться экземпляр — то ли автоматически при его вызове через экземпляр, то ли вручную, когда метод вызывается через класс.

Как было указано во врезке "Как насчет super?" в главе 28, язык Python также располагает встроенной функцией super, которая позволяет вызывать методы суперкласса более обобщенно, но из-за ее недостатков и сложностей мы отложим представление функции super до главы 32. Дополнительные сведения ищите в упомянутой врезке; с этим вызовом связаны известные компромиссы в случае базового применения, а также экзотический расширенный сценарий использования, который для наибольшей эффективности требует универсального развертывания. По причине проблем такого рода предпочтение в книге отдается политике обращения к суперклассам по явному имени, в не посредством встроенной функции super; если вы — новичок в Python, тогда я рекомендую пока применять такой же подход, особенно во время первоначального изучения ООП. Освойте сейчас простой способ, а позже вы сможете сравнить его с другими.

Наследование

Разумеется, весь смысл пространства имен, созданного оператором class, заключается в поддержке наследования имен. В этом разделе мы расширим ряд механизмов и ролей наследования атрибутов в Python.

Как было показано, наследование в Python происходит, когда объект уточняется, и оно инициирует поиск в дереве определения атрибутов — в одном и большем числе пространств имен. Каждый раз, когда вы используете выражение вида *объект.атрибут*, где *объект* представляет собой объект экземпляра или класса, Python осуществляет поиск в дереве пространств имен снизу вверх, начиная с *объекта*, с целью нахождения первого *атрибута*, который удастся обнаружить. Сюда входят ссылки на атрибуты self в ваших методах. Поскольку определения, расположенные ниже в дереве, переопределяют те, что находятся выше, наследование формирует основу специализации.

Построение дерева атрибутов

На рис. 29.1 иллюстрируется способ построения деревьев атрибутов и наполнение их именами. В целом:

- атрибуты экземпляров генерируются операторами присваивания значений атрибутам self в методах;
- атрибуты классов создаются операторами (присваивания) в операторах class;
- ссылки на суперклассы образуются на основе списков классов внутри круглых скобок в заголовке оператора class.

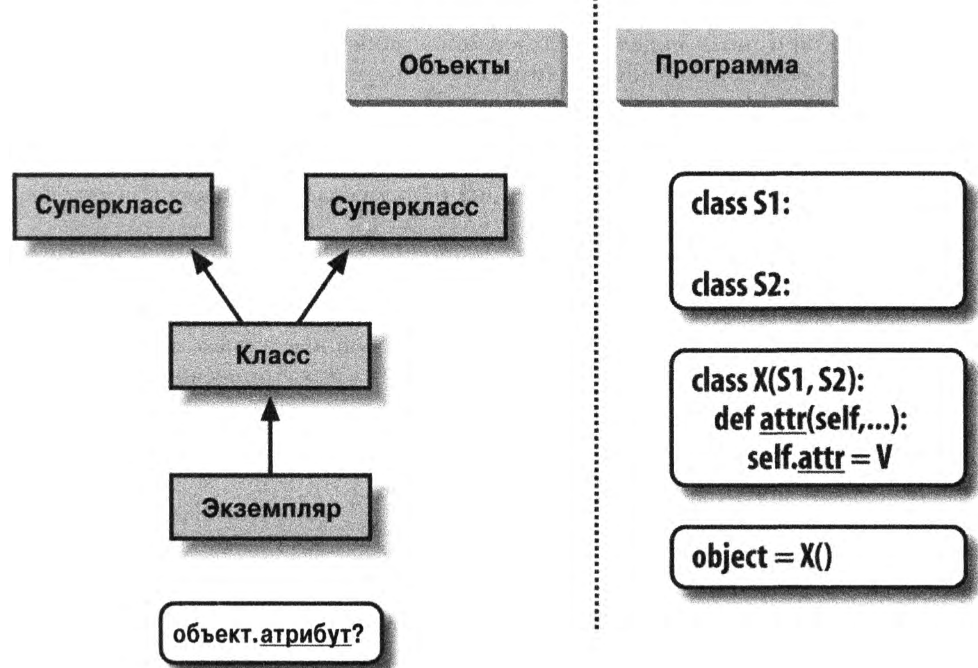


Рис. 29.1. Код программы создает дерево объектов в памяти, где будет происходить поиск со стороны наследования атрибутов. Обращение к классу создает новый экземпляр, который запоминает свой класс, выполнение оператора `class` *создает новый класс, а суперклассы перечисляются внутри круглых скобок в заголовке оператора* `class`*. Каждая ссылка на атрибут запускает новую процедуру восходящего поиска в дереве – даже ссылки на атрибуты* `self` *внутри методов класса*

Конечным результатом оказывается дерево пространств имен атрибутов, ведущее от экземпляра к классу, из которого он был создан, и ко всем суперклассам, перечисленным в заголовке оператора `class`. Всякий раз, когда вы применяете уточнение для извлечения имени атрибута из объекта экземпляра, в этом дереве инициируется восходящий поиск в направлении от экземпляров к суперклассам[4].

Во-вторых, как тоже отмечалось в главе 27, полная история о *наследовании* становится еще более запутанной, когда к смеси добавляются сложные темы наподобие *метаклассов* и *дескрипторов* – и по этой причине мы отложим формальное определение вплоть до главы 40. Однако при обычном использовании наследование представляет собой просто способ переопределения и, следовательно, настройки поведения, реализованного в классах.

[4] Здесь есть две тонкости. Во-первых, приведенное описание не является полным на 100%, потому что мы можем создавать атрибуты экземпляров и классов также путем присваивания им объектов за пределами операторов `class` – но такой подход гораздо менее распространен и более подвержен ошибкам (изменения не изолированы внутри операторов `class`). По умолчанию все атрибуты в Python всегда доступны. Дополнительные сведения о *защите* имен атрибутов от несанкционированного доступа вы найдете в главе 30 при обсуждении `__setattr__`, в главе 31 при изучении имен `__X` и еще раз в главе 39, где мы займемся ее реализацией с помощью декоратора класса.

Специализации унаследованных методов

Только что описанная модель наследования с поиском в дереве является великолепным способом специализации систем. Из-за того, что процесс наследования ищет сначала имена в подклассах до проверки суперклассов, подклассы способны замещать стандартное поведение за счет переопределения атрибутов своих суперклассов. Фактически вы можете строить целые системы в виде иерархий классов и затем их расширять путем добавления новых внешних подклассов, не изменяя существующую логику на месте.

Идея переопределения унаследованных имен положена в основу многообразия методик специализации. Например, подклассы могли бы полностью *замещать* унаследованные атрибуты, *предоставлять* атрибуты, которые ожидает обнаружить суперкласс, и *расширять* методы суперклассов, вызывая их в переопределяемых методах. Мы уже демонстрировали несколько таких схем в действии, а ниже приведен самодостаточный пример расширения:

```
>>> class Super:
        def method(self):
            print('in Super.method')

>>> class Sub(Super):
        def method(self):                   # Переопределить метод
            print('starting Sub.method')    # Добавить здесь нужные действия
            Super.method(self)              # Запустить стандартное действие
            print('ending Sub.method')
```

Вся суть здесь кроется в прямом вызове метода суперкласса. Класс Sub замещает функцию method суперкласса Super собственной специализированной версией, но внутри самого замещения Sub обращается к версии, экспортируемой Super, чтобы обеспечить выполнение стандартного поведения. Другими словами, Sub.method просто расширяет поведение Super.method, а не полностью замещает его:

```
>>> x = Super()                             # Создать экземпляр Super
>>> x.method()                              # Выполняется Super.method
in Super.method
>>> x = Sub()                               # Создать экземпляр Sub
>>> x.method()                              # Выполняется Sub.method,
                                            #    вызывается Super.method
starting Sub.method
in Super.method
ending Sub.method
```

Такая схема реализации расширения часто используется в конструкторах; пример ищите в разделе "Методы" ранее в главе.

Методики связывания классов

Расширение является всего лишь одним способом связывания с суперклассом. В файле specialize.py, обсуждаемом в настоящем разделе, определено множество классов, которые иллюстрируют разнообразные методики.

- Super. Определяет функцию method и метод delegate, который ожидает наличие в подклассе метода action.
- Inheritor. Не предоставляет никаких новых имен, поэтому получает все, что определено в Super.

- Replacer. Переопределяет method из Super посредством собственной версии.
- Extender. Настраивает method из Super, переопределяя и обеспечивая выполнение стандартного поведения.
- Provider. Реализует метод action, ожидаемый методом delegate из Super.

Исследуйте каждый из предложенных классов, чтобы получить представление о различных способах настройки их общего суперкласса. Вот содержимое файла:

```
class Super:
    def method(self):
        print('in Super.method')             # Стандартное поведение
    def delegate(self):
        self.action()                        # Ожидается определение метода

class Inheritor(Super):                      # Буквальное наследование метода
    pass

class Replacer(Super):                       # Полное замещение метода
    def method(self):
        print('in Replacer.method')          # в Replacer.method

class Extender(Super):                       # Расширение поведения метода
    def method(self):
        print('starting Extender.method')    # начало Extender.method
        Super.method(self)
        print('ending Extender.method')      # конец Extender.method

class Provider(Super):                       # Заполнение обязательного метода
    def action(self):
        print('in Provider.action')          # в Provider.method
if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method()
    print('\nProvider...')
    x = Provider()
    x.delegate()
```

Здесь полезно указать на несколько моментов. Прежде всего, обратите внимание на то, как код самотестирования в конце файла specialize.py создает внутри цикла for экземпляры трех разных классов. Поскольку классы являются объектами, вы можете сохранять их в кортеже и создавать экземпляры обобщенным образом без дополнительного синтаксиса (позже идея будет раскрыта более подробно). Подобно модулям классы также имеют специальный атрибут __name__; он заранее устанавливается в строку, содержащую имя из заголовка класса. Вот что происходит после запуска файла:

```
% python specialize.py

Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Абстрактные суперклассы

В рамках классов из предыдущего примера Provider может быть самым важным для понимания. Когда мы вызываем метод delegate через экземпляр Provider, происходят *два* независимых поиска со стороны процедуры наследования.

1. В начальном вызове x.delegate интерпретатор Python находит метод delegate в Super, выполняя поиск в экземпляре Provider и выше. Экземпляр x передается в аргументе self метода как обычно.
2. Внутри метода Super.delegate метод self.action инициирует новый независимый поиск в self и выше. Из-за того, что self ссылается на экземпляр Provider, метод action обнаруживается в подклассе Provider.

Такая кодовая структура вида "заполнение пропусков" типична для объектно-ориентированных фреймворков. В более реалистичном контексте заполняемый подобным образом метод мог бы обрабатывать какое-то событие в графическом пользовательском интерфейсе, снабжать данными для визуализации как часть веб-страницы, анализировать текст дескриптора в XML-файле и т.д. — ваш подкласс предоставляет специфические действия, но оставшуюся часть всей работы выполняет фреймворк.

По крайней мере, в свете метода delegate суперкласс в этом примере является тем, что иногда называют *абстрактным суперклассом* — классом, который ожидает от своих подклассов предоставления частей своего поведения. Если ожидаемый метод в подклассе не определен, тогда после неудавшегося поиска при наследовании Python генерирует исключение неопределенного имени.

Временами создатели классов делают такие требования к подклассам более очевидными с помощью операторов assert или путем генерации встроенного исключения NotImplementedError посредством операторов raise. В следующей части книги мы подробнее исследуем операторы, способные генерировать исключения; для краткого предварительного ознакомления ниже показана схема assert в действии:

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!'    # Если вызвана эта версия

>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
AssertionError: метод action должен быть определен!
```

Мы рассмотрим оператор assert в главах 33 и 34; говоря кратко, если результатом вычисления его первого выражения будет False, тогда он генерирует исключение с указанным сообщением об ошибке. Здесь выражение всегда имеет результат False, чтобы инициировать выдачу сообщения об ошибке, когда метод не переопределен, и процесс наследования находит данную версию. В качестве альтернативы некоторые классы в таких методах-заглушках просто генерируют исключение NotImplementedError, сигнализируя об ошибке:

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        raise NotImplementedError('action must be defined!')
```

```
>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!
NotImplementedError: метод action должен быть определен!
```

Для экземпляров подклассов мы по-прежнему получаем исключение до тех пор, пока подкласс не предоставит ожидаемый метод для замещения стандартной версии в суперклассе:

```
>>> class Sub(Super): pass

>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!

>>> class Sub(Super):
        def action(self): print('spam')

>>> X = Sub()
>>> X.delegate()
spam
```

Упражнение 8 (систематизация животных в зоопарке) в конце главы 32 посвящено более реалистичному примеру реализации концепций, обсуждаемых в настоящем разделе (решение упражнения приведено в приложении Г). Такая систематизация представляет собой традиционный способ введения в ООП, хотя она лежит несколько в стороне от служебных обязанностей большинства разработчиков (приношу извинения тем читателям, кто по стечению обстоятельств работает в зоопарке!).

Абстрактные суперклассы в Python 3.X и Python 2.6+: обзор

С выходом версий Python 2.6 и 3.0 абстрактные суперклассы из предыдущего раздела (они же "абстрактные базовые классы"), которые требуют заполнения методов подклассами, также могут быть реализованы посредством специального синтаксиса классов. Способ написания кода слегка варьируется в зависимости от версии. В Python 3.X мы применяем ключевой аргумент в заголовке оператора `class` вместе со специальным декораторным синтаксисом `@`, которые будут более подробно описаны далее в книге:

```
from abc import ABCMeta, abstractmethod

class Super(metaclass=ABCMeta):
    @abstractmethod
    def method(self, ...):
        pass
```

Но в Python 2.6 и 2.7 взамен мы используем атрибут класса:

```
class Super:
    __metaclass__ = ABCMeta
    @abstractmethod
    def method(self, ...):
        pass
```

В обоих случаях результат тот же — мы не можем создать экземпляр, пока не определим метод ниже в дереве классов. Скажем, для примера из предыдущего раздела в Python 3.X предусмотрен специальный эквивалентный синтаксис:

```
>>> from abc import ABCMeta, abstractmethod
>>>
```

```
>>> class Super(metaclass=ABCMeta):
        def delegate(self):
            self.action()
        @abstractmethod
        def action(self):
            pass

>>> X = Super()
TypeError: Can't instantiate abstract class Super with abstract methods action
TypeError: невозможно создать экземпляр абстрактного класса Super с
абстрактными методами action

>>> class Sub(Super): pass

>>> X = Sub()
TypeError: Can't instantiate abstract class Sub with abstract methods action
TypeError: невозможно создать экземпляр абстрактного класса Sub с
абстрактными методами action

>>> class Sub(Super):
        def action(self): print('spam')

>>> X = Sub()
>>> X.delegate()
spam
```

При такой реализации создавать экземпляры класса с абстрактным методом нельзя (т.е. мы не можем создать экземпляр, обратившись к нему), пока в подклассах не будут определены все его абстрактные методы. Хотя такой подход требует написания большего объема кода и добавочных знаний, его потенциальное преимущество в том, что сообщения об ошибках, связанных с недостающими методами, выдаются при попытке создания экземпляра класса, а не позже, когда мы пытаемся вызвать отсутствующий метод. Это средство можно применять и для определения ожидаемого интерфейса, автоматически проверяемого в клиентских классах.

К сожалению, продемонстрированная схема также опирается на два расширенных языковых инструмента, с которыми мы пока еще не сталкивались — *декораторы функций*, представляемые в главе 32 и подробно рассматриваемые в главе 39, плюс *объявления метаклассов*, которые упоминаются в главе 32 и раскрываются в главе 40 — поэтому здесь мы не будем касаться остальных аспектов данного средства. Дополнительные сведения ищите в стандартных руководствах по Python, а также в готовых абстрактных суперклассах, предлагаемых Python.

Пространства имен: заключение

Итак, после того, как мы исследовали объекты классов и экземпляров, история пространств имен Python завершена. В справочных целях ниже приведена краткая сводка всех правил, применяемых для распознавания имен. Первое, что следует запомнить — уточненные и неуточненные имена трактуются по-разному, а некоторые области видимости предназначены для инициализации пространств имен объектов:

- неуточненные имена (например, X) имеют дело с областями видимости;
- уточненные имена (скажем, `object.X`) используют пространства имен объектов;
- определенные области видимости инициализируют пространства имен объектов (для модулей и классов).

Иногда эти концепции взаимодействуют — например, в `object.X` имя `object` ищется по областям видимости, после чего в результирующих объектах производится поиск имени `X`. Поскольку области видимости и пространства имен жизненно важны для понимания кода на Python, давайте подведем более детальные итоги по правилам.

Простые имена: глобальные, если не выполнено их присваивание

Как уже известно, неуточненные простые имена следуют правилу лексических областей видимости LEGB, кратко описанному при исследовании функций в главе 17 первого тома.

Присваивание *(`X = значение`)*

По умолчанию делает имя локальным: создает либо изменяет имя `X` в текущей локальной области видимости, если только оно не объявлено как `global` (или `nonlocal` в Python 3.X).

Ссылка *(`X`)*

Ищет имя `X` в текущей локальной области видимости, затем в любых объемлющих функциях, далее в текущей глобальной области видимости, затем во встроенной области видимости согласно правилу LEGB. Поиск во включающих классах не выполняется: взамен имена классов извлекаются как атрибуты объектов.

Кроме того, как упоминалось в главе 17 первого тома, некоторые конструкции в особых сценариях дополнительно локализуют имена (например, переменные в ряде включений и конструкции операторов `try`), но подавляющее большинство имен следуют правилу LEGB.

Имена атрибутов: пространства имен объектов

Мы также видели, что уточненные имена атрибутов ссылаются на атрибуты специфических объектов и подчиняются правилам для модулей и классов. В случае объектов классов и экземпляров правила ссылки дополняются с целью включения процедуры поиска при наследовании.

Присваивание *(`object.X = значение`)*

Создает или модифицирует имя атрибута `X` в пространстве имен уточняемого объекта `object` и больше нигде. Подъем по дереву наследования происходит только при ссылке на атрибут, но не в случае присваивания значения атрибуту.

Ссылка *(`object.X`)*

Для объектов, основанных на классах, ищет имя атрибута `X` в `object` и затем во всех доступных классах выше, применяя процедуру поиска при наследовании. Для объектов, не основанных на классах, таких как модули, извлекает `X` из `object` напрямую.

Как упоминалось ранее, предшествующие правила отражают нормальный и типичный сценарий. Правила распознавания имен атрибутов могут варьироваться в классах, которые используют более развитые инструменты, особенно в классах нового стиля — вариант в Python 2.X и стандарт в Python 3.X, которые мы будем исследовать в

главе 32. Скажем, ссылочное наследование может обладать более широкими возможностями, чем здесь предполагается, когда развернуты метаклассы, а классы, которые задействуют инструменты управления атрибутами, такие как свойства, дескрипторы и `__setattr__`, могут перехватывать и направлять присваивания значений атрибутам произвольным образом.

На самом деле определенная процедура наследования *запускается* и при присваивании для нахождения дескрипторов с методом `__set__` в классах нового стиля; такие инструменты переопределяют нормальные правила для ссылки и присваивания. В главе 38 мы подробно обсудим инструменты управления атрибутами, а в главе 40 формализуем наследование и то, как оно применяет дескрипторы. Пока что большинство читателей должны сосредоточить внимание на приведенных здесь нормальных правилах, которые охватывают бо́льшую часть прикладного кода на Python.

"Дзен" пространств имен: присваивания классифицируют имена

Из-за отличающихся процедур поиска для уточненных и неуточненных имен и множества уровней просмотра в обеих процедурах временами трудно сказать, где в итоге будет найдено то или иное имя. В Python критически важно место, где имени *присваивается* значение — место полностью определяет область видимости или объект, в котором имя будет находиться. В файле `manynames.py`, содержимое которого показано ниже, демонстрируется перевод этого принципа в код, а также подытоживаются идеи пространств имен, встречающиеся повсеместно в книге (без учета не особо ясных областей видимости особых сценариев вроде включений):

```
# Файл manynames.py
X = 11                 # Глобальное имя/атрибут модуля (X или manynames.X)

def f():
    print(X)           # Доступ к глобальному имени X (11)

def g():
    X = 22             # Локальная переменная в функции (X, скрывает X в модуле)
    print(X)

class C:
    X = 33             # Атрибут класса (C.X)
    def m(self):
        X = 44         # Локальная переменная в методе (X)
        self.X = 55    # Атрибут экземпляра (экземпляр.X)
```

В файле одно и то же имя `X` присваивается пять раз — иллюстративная, хотя не совсем лучшая практика! Тем не менее, поскольку это имя присваивается в пяти разных местах, все пять `X` в программе являются совершенно разными переменными. Просматривая сверху вниз, вот что генерируют присваивания `X`: атрибут модуля (`11`), локальную переменную в функции (`22`), атрибут класса (`33`), локальную переменную в методе (`44`) и атрибут экземпляра (`55`). Хотя все пять имеют имя `X`, тот факт, что всем им присваивались значения в разных местах исходного кода, или то, что им были присвоены разные объекты, делает их всех уникальными переменными.

Вы должны посвятить время внимательному изучению примера, т.к. в нем собраны идеи, которые исследовались в нескольких последних частях книги. Осмыслив его должным образом, вы обретете знания пространств имен Python. Или можете запустить код и посмотреть, что произойдет. Вот остаток файла исходного кода, где создается экземпляр, а также и выводятся все переменные `X`, которые можно извлекать:

```
# Файл manynames.py, продолжение

if __name__ == '__main__':
    print(X)                  # 11: имя из модуля (оно же manynames.X
                              #     за пределами файла)
    f()                       # 11: глобальное имя
    g()                       # 22: локальное имя
    print(X)                  # 11: имя из модуля не изменилось

    obj = C()                 # Создать экземпляр
    print(obj.X)              # 33: имя из класса, унаследованное экземпляром

    obj.m()                   # Присоединения имени атрибута X к экземпляру
    print(obj.X)              # 55: имя из экземпляра
    print(C.X)                # 33: имя из класса (оно же obj.X, если X
                              #     в экземпляре отсутствует)

    #print(C.m.X)             # НЕУДАЧА: видимо только в методе
    #print(g.X)               # НЕУДАЧА: видимо только в функции
```

Вывод, полученный в результате запуска файла, указан в комментариях внутри кода; отследите их, чтобы посмотреть, как каждый раз осуществляется доступ к переменной по имени X. В частности обратите внимание, что мы можем указывать класс для извлечения его атрибута (C.X), но извлечь локальные переменные внутри функций или методов извне их операторов def не удастся. Локальные переменные видимы только другому коду внутри def, и в действительности они существуют в памяти только в период, пока выполняется вызов функции или метода.

Некоторые имена, определенные в этом файле, видимы также *за пределами файла* другим модулям, но не забывайте, что мы обязаны всегда импортировать файл, прежде чем сможем иметь доступ к его именам — в конце концов, как раз изоляция имен является сущностью модулей:

```
# Файл otherfile.py

import manynames

X = 66
print(X)                     # 66: здесь глобальное имя
print(manynames.X)           # 11: после импортирования глобальные имена
становятся атрибутами

manynames.f()                # 11: имя X из модуля manynames, не то, которое
находится здесь!
manynames.g()                # 22: локальное имя из функции в другом файле

print(manynames.C.X)         # 33: атрибут класса в другом модуле
I = manynames.C()
print(I.X)                   # 33: здесь все еще имя из класса
I.m()
print(I.X)                   # 55: а теперь имя из экземпляра!
```

Обратите внимание, что вызов manynames.f() выводит X из manynames, не имя X, присвоенное в данном файле — области видимости всегда определяются местоположением присваиваний в исходном коде (т.е. лексически) и никогда не зависят от того, что и куда импортируется. Кроме того, имейте в виду, что собственное имя X экземпляра не создается до тех пор, пока мы не вызовем I.m() — подобно всем переменным атрибуты появляются, когда им присваиваются значения, не ранее. Обычно мы создаем атрибуты экземпляров путем присваивания им значений в методах конструкторов __init__ классов, но это не единственный вариант.

Наконец, как известно из главы 17 первого тома, функция в состоянии *изменять* имена за пределами самой себя с помощью операторов `global` и (в Python 3.X) `nonlocal` – указанные операторы предоставляют доступ по записи, но также модифицируют правила привязки присваивания к пространству имен:

```
X = 11                # Глобальное имя в модуле

def g1():
    print(X)          # Ссылка на глобальное имя в модуле (11)

def g2():
    global X
    X = 22            # Изменение глобального имени в модуле

def h1():
    X = 33            # Локальное имя в функции
    def nested():
        print(X)      # Ссылка на локальное имя в объемлющей области видимости (33)

def h2():
    X = 33            # Локальное имя в функции
    def nested():
        nonlocal X    # Оператор Python 3.X
        X = 44        # Изменение локального имени в объемлющей области видимости
```

Разумеется, в общем случае вы не должны использовать одно и то же имя для всех переменных в сценарии. Однако, как было продемонстрировано в примере, даже если вы поступите подобным образом, то пространства имен Python будут работать так, чтобы предотвратить непредумышленные конфликты между именами, применяемыми в одном контексте, и именами, задействованными в другом.

Вложенные классы: снова о правиле областей видимости LEGB

В предыдущем примере резюмировалось влияние на области видимости вложенных функций, которые обсуждалось в главе 17 первого тома. Оказывается, что классы тоже можно вкладывать – полезная кодовая схема в определенных видах программ, с последствиями в отношении областей видимости, которые естественным образом вытекают из имеющихся у вас знаний, но на первый взгляд могут показаться неочевидными. В этом разделе концепция иллюстрируется на примере.

Несмотря на то что классы обычно определяются на верхнем уровне модуля, иногда они являются вложенными в генерирующие их функции – вариация на тему "фабричных функций" (*замыканий*) из главы 17 первого тома с похожими ролями сохранения состояния. Там мы отмечали, что операторы `class` вводят новые локальные области видимости во многом подобно операторам `def` функций, которые следуют тому же самому правилу LEGB, как и определения функций.

Правило LEGB применяется к верхнему уровню самого класса и к верхнему уровню вложенных в него функций методов. Оба формируют уровень *L* этого правила – они представляют собой нормальные локальные области видимости с доступом к их именам, именам в любых объемлющих функциях, глобальным именам во включающем модуле и встроенной области видимости. Как и модули, после выполнения оператора `class` локальная область видимости класса *превращается* в пространство имен атрибутов.

Хотя классы имеют доступ к областям видимости объемлющих функций, они не действуют в качестве объемлющих областей видимости для кода, вложенного внутрь

класса: Python ищет имена, на которые произведена ссылка, в объемлющих функциях, но *никогда* не выполняет их поиск в объемлющих классах. То есть класс *является* локальной областью видимости и имеет доступ к объемлющим локальным областям видимости, но он не *служит* объемлющей локальной областью видимости для дальнейшего вложенного кода. Из-за того, что процесс поиска имен, используемых в функциях методов, пропускает включающий класс, атрибуты класса должны извлекаться как атрибуты объекта с применением наследования.

Например, в следующей функции nester все ссылки на X направляются в глобальную область видимости кроме последней, которая подбирает переопределение из локальной области видимости (данный раздел кода находится в файле classscope.py, а вывод каждого примера описан в последних двух комментариях):

```
X = 1

def nester():
    print(X)                # Глобальное имя: 1
    class C:
        print(X)            # Глобальное имя: 1
        def method1(self):
            print(X)        # Глобальное имя: 1
        def method2(self):
            X = 3           # Скрывает глобальное имя
            print(X)        # Локальное имя: 3
    I = C()
    I.method1()
    I.method2()

print(X)                    # Глобальное имя: 1
nester()                    # Остаток: 1, 1, 1, 3
print('-'*40)
```

Тем не менее, взгляните, что происходит, когда мы заново присваиваем значение тому же самому имени на уровнях вложенных функций. Переопределение X создает локальные имена, которые скрывают имена из объемлющих областей видимости, в точности как для простых вложенных функций; уровень включающего класса вовсе не изменяет это правило и на самом деле не имеет к нему отношения:

```
X = 1

def nester():
    X = 2                   # Скрывает глобальное имя
    print(X)                # Локальное имя: 2
    class C:
        print(X)            # В объемлющем def (nester): 2
        def method1(self):
            print(X)        # В объемлющем def (nester): 2
        def method2(self):
            X = 3           # Скрывает имя из объемлющего def (nester)
            print(X)        # Локальное имя: 3
    I = C()
    I.method1()
    I.method2()

print(X)                    # Глобальное имя: 1
nester()                    # Остаток: 2, 2, 2, 3
print('-'*40)
```

А вот что случается, когда мы заново присваиваем значение тому же самому имени несколько раз по пути: присваивания в локальных областях видимости функций и классов скрывают глобальные имена или совпадающие локальные имена из объемлющих функций независимо от вложенности:

```
X = 1

def nester():
    X = 2                  # Скрывает глобальное имя
    print(X)               # Локальное имя: 2
    class C:
        X = 3              # Локальное имя из класса скрывает имя из nester:
C.X или I.X
        print(X)           # Локальное имя: 3
        def method1(self):
            print(X)       # В объемлющем def (не 3 в классе!): 2
            print(self.X)  # Унаследованное локальное имя класса: 3
        def method2(self):
            X = 4          # Скрывает имя из объемлющей области видимости
(nester, не класса)
            print(X)       # Локальное имя: 4
            self.X = 5     # Скрывает имя из класса
            print(self.X)  # Находится в экземпляре: 5
    I = C()
    I.method1()
    I.method2()

print(X)                   # Глобальное имя: 1
nester()                   # Остаток: 2, 3, 2, 3, 4, 5
print('-'*40)
```

Самое главное, правила поиска для простых имен вроде X никогда не ищут во включающих операторах class — только в операторах def, модулях и встроенной области видимости (правило называется LEGB, а не CLEGB!). Скажем, в method1 имя X находится в def за пределами включающего класса, который имеет то же самое имя в своей локальной области видимости. Чтобы получить имена, присвоенные в классе (например, методы), мы обязаны извлекать их как атрибуты объекта класса или экземпляра через self.X в данном случае.

Хотите верьте, хотите нет, но позже в книге мы увидим сценарии применения для такой кодовой схемы с вложенными классами, особенно в ряде *декораторов* в главе 39. В этой роли объемлющая функция обычно служит фабрикой классов и предоставляет сохраненное состояние для последующего использования во включающем классе или в его методах.

Словари пространств имен: обзор

В главе 23 первого тома мы выяснили, что пространства имен модулей имеют конкретную реализацию в виде словарей, доступную через встроенный атрибут __dict__. В главах 27 и 28 мы узнали, что то же самое остается справедливым для объектов классов и экземпляров — уточнение атрибутов внутренне обычно представляет собой операцию индексирования в словаре. На самом деле объекты классов и экземпляров в Python по большей части являются всего лишь словарями со связями между ними. Доступ к таким словарям, равно как к их связям, обеспечивается для участия в расширенных ролях (скажем, для создания инструментов).

Мы применяли ряд инструментов подобного рода в предыдущей главе, но чтобы подвести итог и помочь вам лучше понять, каким образом внутренне работают атрибуты, давайте запустим интерактивный сеанс, который отслеживает рост словарей пространств имен, когда задействованы классы. Теперь, обладая дополнительными знаниями о методах и суперклассах, мы можем предоставить здесь улучшенный обзор. Первым делом определим суперкласс и подкласс с методами, которые будут сохранять данные в своих экземплярах:

```
>>> class Super:
        def hello(self):
            self.data1 = 'spam'

>>> class Sub(Super):
        def hola(self):
            self.data2 = 'eggs'
```

Когда мы создаем экземпляр подкласса, он начинает свое существование с пустого словаря пространства имен, но имеет связи с классом для обеспечения работы поиска при наследовании. В действительности дерево наследования явно доступно в специальных атрибутах, которые можно исследовать. Экземпляры располагают атрибутом `__class__`, связывающим их с классом, а классы имеют атрибут `__bases__`, который представляет собой кортеж, содержащий связи с находящимися выше в дереве суперклассами (приведенный далее код запускался в Python 3.7; ваши форматы имен, внутренние атрибуты и порядок ключей могут отличаться):

```
>>> X = Sub()
>>> X.__dict__                    # Словарь пространства имен экземпляра
{}
>>> X.__class__                   # Класс экземпляра
<class '__main__.Sub'>
>>> Sub.__bases__                 # Суперкласс класса
(<class '__main__.Super'>,)
>>> Super.__bases__               # Пустой кортеж () в Python 2.X
(<class 'object'>,)
```

Когда классы выполняют присваивание атрибутам `self`, они заполняют объекты экземпляров — т.е. атрибуты оказываются в словарях пространств имен экземпляров, а не классов. Пространство имен объекта экземпляра записывает данные, которые могут варьироваться от экземпляра к экземпляру, причем `self` является привязкой к этому пространству имен:

```
>>> Y = Sub()

>>> X.hello()
>>> X.__dict__
{'data1': 'spam'}

>>> X.hola()
>>> X.__dict__
{'data1': 'spam', 'data2': 'eggs'}

>>> list(Sub.__dict__.keys())
['__module__', 'hola', '__doc__']
>>> list(Super.__dict__.keys())
['__module__', 'hello', '__dict__', '__weakref__', '__doc__']

>>> Y.__dict__
{}
```

Обратите внимание на добавочные имена с символами подчеркивания в словарях классов; Python устанавливает такие имена автоматически, и мы можем отфильтровать их с помощью генераторных выражений, которые были показаны в главах 27 и 28, но здесь не повторяются. Большинство таких имен в обычных программах не используется, но есть инструменты, применяющие некоторые из них (например, в `__doc__` хранятся строки документации, обсуждаемые в главе 15 первого тома).

Также обратите внимание, что `Y`, второй экземпляр, созданный в начале сеанса, в конце по-прежнему имеет пустой словарь пространства имен, хотя словарь экземпляра `X` заполнялся присваиваниями в методах. Опять-таки каждый экземпляр располагает независимым словарем пространства имен, пустым в самом начале и способным хранить атрибуты, которые полностью отличаются от атрибутов, записанных в словарях пространств имен других экземпляров того же самого класса.

Поскольку атрибуты в действительности представляют собой ключи словаря внутри Python, на самом деле существуют два способа извлечения и присваивания им значений — уточнение либо индексирование по ключу:

```
>>> X.data1, X.__dict__['data1']
('spam', 'spam')

>>> X.data3 = 'toast'
>>> X.__dict__
{'data1': 'spam', 'data2': 'eggs', 'data3': 'toast'}

>>> X.__dict__['data3'] = 'ham'
>>> X.data3
'ham'
```

Однако такая эквивалентность применима только к атрибутам, фактически присоединенным к *экземпляру*. Так как уточнение при извлечении атрибутов тоже выполняет поиск в дереве наследования, оно может получать доступ к *унаследованным* атрибутам, что невозможно сделать через индексирование в словаре пространства имен. Скажем, к унаследованному атрибуту `X.hello` нельзя обратиться посредством `X.__dict__['hello']`.

Поэкспериментируйте с этими специальными атрибутами самостоятельно, чтобы получить лучшее представление о том, как пространства имен действительно работают с атрибутами. Кроме того, попробуйте прогнать имеющиеся объекты через функцию `dir`, с которой мы встречались в предшествующих двух главах — вызов `dir(X)` подобен `X.__dict__.keys()`, но помимо сортировки своего списка функция `dir` также включает ряд унаследованных и встроенных атрибутов. Даже если вам никогда не придется использовать все это при разработке собственных программ, знание того, что они являются всего лишь нормальными словарями, может содействовать пониманию пространств имен в целом.

В главе 32 мы также узнаем о *слотах* — усовершенствованной возможности классов нового стиля хранить атрибуты в экземплярах, но не в их словарях пространств имен. Их заманчиво трактовать как атрибуты классов, и они на самом деле появляются в пространствах имен, где управляют значениями для каждого экземпляра. Тем не менее, как мы увидим, слоты могут полностью воспрепятствовать созданию `__dict__` в экземпляре — потенциальная ситуация, возникновение которой обобщенные инструменты должны иногда учитывать за счет применения нейтральных к хранению средств, таких как `dir` и `getattr`.

Связи между пространствами имен: инструмент подъема по дереву

В предыдущем разделе демонстрировались специальные атрибуты экземпляров и классов `__class__` и `__bases__` без реального объяснения, почему они могут вообще интересовать. Формулируя кратко, упомянутые атрибуты позволяют инспектировать иерархии наследования внутри написанного вами кода. Например, их можно использовать для отображения дерева классов, как показано ниже в коде на Python 3.X и 2.X:

```
#!python
"""
classtree.py: подъем по деревьям наследования с применением связей между
пространствами имен и отображением находящихся выше суперклассов с отступом
согласно высоте
"""

def classtree(cls, indent):
    print('.' * indent + cls.__name__)    # Вывести здесь имя класса
    for supercls in cls.__bases__:        # Вызвать рекурсивно для всех
                                          #   суперклассов
        classtree(supercls, indent+3)     # Может посетить суперкласс
                                          #   более одного раза

def instancetree(inst):
    print('Tree of %s' % inst)            # Показать экземпляр
    classtree(inst.__class__, 3)          # Подняться к его классу

def selftest():
    class A:     pass
    class B(A):  pass
    class C(A):  pass
    class D(B,C): pass
    class E:     pass
    class F(D,E): pass
    instancetree(B())
    instancetree(F())

if __name__ == '__main__': selftest()
```

Функция `classtree` в этом сценарии *рекурсивна* — она выводит имя класса с использованием `__name__` и затем поднимается к его суперклассам, вызывая саму себя. Такая реализация позволяет функции обходить деревья классов произвольной формы; рекурсия поднимается доверху и останавливается на корневых суперклассах, которые имеют пустые атрибуты `__bases__`. Когда применяется рекурсия, каждый активный уровень функции получает собственную копию локальной области видимости; здесь это означает, что аргументы `cls` и `indent` будут разными на каждом уровне `classtree`.

Большую часть файла занимает код самотестирования. При автономном запуске в Python 2.X он строит пустое дерево классов, создает два его экземпляра и выводит структуры их деревьев классов:

```
C:\code> c:\python27\python classtree.py
Tree of <__main__.B instance at 0x00000000022C3A88>
...B
......A
```

```
Tree of <__main__.F instance at 0x00000000022C3A88>
...F
......D
.........B
............A
.........C
............A
......E
```

В случае запуска в Python 3.X дерево включает подразумеваемый суперкласс `object`, который автоматически добавляется над автономными корневыми (т.е. самыми верхними) классами, потому что в Python 3.X все классы являются классами "нового стиля" – более подробно о таком изменении речь пойдет в главе 32:

```
C:\code> c:\python37\python classtree.py
Tree of <__main__.selftest.<locals>.B object at 0x00000000012BC4A8>
...B
......A
.........object
Tree of <__main__.selftest.<locals>.F object at 0x00000000012BC4A8>
...F
......D
.........B
............A
...............object
.........C
............A
...............object
......E
.........object
```

Отступы, отмечаемые точками, используются для обозначения высоты дерева классов. Конечно, мы могли бы улучшить формат вывода и возможно даже изобразить его в графическом пользовательском интерфейсе. Однако и в существующем виде мы в состоянии импортировать эти функции везде, где желательно быстро отобразить дерево классов:

```
C:\code> c:\python37\python
>>> class Emp: pass

>>> class Person(Emp): pass

>>> bob = Person()

>>> import classtree
>>> classtree.instancetree(bob)
Tree of <__main__.Person object at 0x0000000002E8DC88>
...Person
......Emp
.........object
```

Независимо от того, придется ли вам когда-нибудь создавать или применять такие инструменты, в примере был продемонстрирован один из многих способов, которыми можно задействовать специальные атрибуты, открывающие доступ к внутренностям интерпретатора. Вы увидите еще один способ при реализации универсальных инструментов отображения классов `lister.py` в разделе "Множественное наследование: 'подмешиваемые' классы" главы 31 – там мы расширим эту методику, чтобы

также отображать атрибуты каждого объекта в дереве классов и функционировать в качестве общего суперкласса.

В последней части книги мы снова возвратимся к таким инструментам в контексте построения инструментов Python в целом для создания инструментов, которые обеспечивают защиту атрибутов, проверку достоверности аргументов и т.д. Доступ к внутренностям интерпретатора дает возможность реализовывать мощные инструменты разработки, хотя требуется далеко не каждому программисту на Python.

Снова о строках документации

В примере из предыдущего раздела присутствует строка документации для его модуля, но не забывайте, что строки документации можно использовать также для компонентов классов. Строки документации, подробно рассмотренные в главе 15 первого тома, представляют собой строковые литералы, которые отображаются в верхней части разнообразных структур и автоматически сохраняются Python в атрибутах `__doc__` соответствующих объектов. Они работают для файлов модулей, операторов `def` определения функций, а также классов и методов.

Теперь, лучше зная классы и методы, в файле `docstr.py`, содержимое которого показано ниже, предлагается короткий, но исчерпывающий пример, демонстрирующий места, где допускается появление строк документации в коде. Все они могут быть блоками в утроенных кавычках или более простыми однострочными литералами вроде приведенных далее:

```
"I am: docstr.__doc__"

def func(args):
    "I am: docstr.func.__doc__"
    pass

class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__ or self.__doc__"
    def method(self):
        "I am: spam.method.__doc__ or self.method.__doc__"
        print(self.__doc__)
        print(self.method.__doc__)
```

Основное преимущество строк документации заключается в том, что они не исчезают во время выполнения. Соответственно, если для объекта была предусмотрена строка документации, тогда вы можете дополнить объект атрибутом `__doc__` и извлечь его документацию (символы разрыва строки должным образом интерпретируются при выводе):

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__doc__'
>>> docstr.func.__doc__
'I am: docstr.func.__doc__'
>>> docstr.spam.__doc__
'I am: spam.__doc__ or docstr.spam.__doc__ or self.__doc__'
>>> docstr.spam.method.__doc__
'I am: spam.method.__doc__ or self.method.__doc__'

>>> x = docstr.spam()
>>> x.method()
I am: spam.__doc__ or docstr.spam.__doc__ or self.__doc__
I am: spam.method.__doc__ or self.method.__doc__
```

В главе 15 первого тома обсуждался инструмент *PyDoc*, которому известно, как форматировать все эти строки в отчетах и на веб-страницах. Ниже представлен результат выполнения функции `help` для `docstr` в Python 3.7:

```
>>> help(docstr)
Help on module docstr:

NAME
    docstr - I am: docstr.__doc__

CLASSES
    builtins.object
        spam

    class spam(builtins.object)
     |  I am: spam.__doc__ or docstr.spam.__doc__ or self.__doc__
     |
     |  Methods defined here:
     |
     |  method(self)
     |      I am: spam.method.__doc__ or self.method.__doc__
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    func(args)
        I am: docstr.func.__doc__

FILE
    c:\code\docstr.py
```

Строки документации доступны во время выполнения, но синтаксически они менее гибкие, чем комментарии #, которые могут появляться в программе где угодно. Обе формы являются полезными инструментами, и наличие любой документации по программе – хороший факт (разумеется, при условии, что она точна!). Как утверждалось ранее, эмпирическое правило "рекомендуемой методики" в Python предполагает применение строк документации для документирования функциональности (что объекты делают) и комментариев # для документирования на микро-уровнях (как работают загадочные порции кода).

Классы или модули

Наконец, давайте завершим главу кратким сравнением того, что было темами последних двух частей: модулей и классов. Поскольку и модули, и классы являются пространствами имен, различие между ними может сбивать с толку.

- Модули:
- реализуют пакеты данных/логики;

- создаются с помощью файлов с кодом на Python или расширений на других языках;
- используются путем импортирования;
- формируют верхний уровень структуры программы на Python.
- Классы:
- реализуют новые полнофункциональные объекты;
- создаются посредством операторов `class`;
- используются путем обращения к ним;
- всегда находятся внутри модуля.

Классы также поддерживают дополнительные возможности, отсутствующие у модулей, такие как перегрузка операций, создание множества экземпляров и наследование. Хотя классы и модули представляют собой пространства имен, вы должны уже понимать, что они являются очень разными вещами. Нам необходимо двигаться вперед, чтобы увидеть, насколько разными могут быть сами классы.

Резюме

В главе мы провели второй, углубленный экскурс в механизмы ООП на языке Python. Вы больше узнали о классах, методах и наследовании. Вдобавок мы завершили историю о пространствах имен и областях видимости в Python, расширив ее применительно к классам. Попутно мы взглянули на несколько более развитых концепций вроде абстрактных суперклассов, атрибутов данных классов, словарей и связей пространств имен, а также ручных вызовов методов и конструкторов суперкласса.

После выяснения механики создания классов на языке Python в главе 30 мы перейдем к специфическому аспекту этой механики: *перегрузке операций*. Затем мы исследуем распространенные паттерны проектирования и рассмотрим ряд способов, которыми обычно классы используются и объединяются для оптимизации многократного применения кода. Но сначала закрепите пройденный материал главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Что такое абстрактный суперкласс?
2. Что происходит, когда простое присваивание появляется на верхнем уровне оператора `class`?
3. Почему в классе может возникнуть потребность в ручном вызове метода `__init__` суперкласса?
4. Как можно дополнить унаследованный метод, не замещая его полностью?
5. Чем локальная область видимости класса отличается от локальной области видимости функции?
6. Какой город был столицей Ассирии?

Проверьте свои знания: ответы

1. Абстрактный суперкласс – это класс, который вызывает метод, но не наследует и не определяет его; он ожидает заполнения метода подклассом. Абстрактные суперклассы часто используются в качестве способа обобщения классов, когда поведение не может быть спрогнозировано до написания кода более специфического подкласса. Объектно-ориентированные фреймворки также применяют их как способ направления на определяемые клиентом настраиваемые операции.

2. Когда простой оператор присваивания (`X = Y`) появляется на верхнем уровне оператора `class`, он присоединяет к классу атрибут данных (`Класс.X`). Подобно всем атрибутам класса он будет разделяться всеми экземплярами; тем не менее, атрибуты данных не являются вызываемыми функциями методов.

3. В классе должен вручную вызываться метод `__init__` суперкласса, если в нем определяется собственный конструктор `__init__` и нужно, чтобы код конструктора суперкласса по-прежнему выполнялся. Сам интерпретатор Python автоматически запускает только один конструктор – самый нижний в дереве. Конструктор суперкласса обычно вызывается через имя класса с передачей экземпляра `self` вручную: `Суперкласс.__init__(self, ...)`.

4. Чтобы вместо замещения дополнить унаследованный метод, его понадобится повторно определить в подклассе, но внутри этой новой версии вручную вызвать версию метода из суперкласса с передачей ей экземпляра `self`: `Суперкласс.метод(self, ...)`.

5. Класс представляет собой локальную область видимости и имеет доступ к объемлющим локальным областям видимости, но он не служит в качестве локальной области видимости для добавочного вложенного кода. Подобно модулям после выполнения оператора `class` локальная область видимости класса превращается в пространство имен атрибутов.

6. Ашшур (или Калат-Шергат), Калах (или Нимруд), короткое время Дур-Шаррукин (или Хорсабад) и в заключение Ниневия.

ГЛАВА 30

Перегрузка операций

В этой главе продолжается доскональное исследование механики классов с переключением внимания на перегрузку операций. Мы кратко затрагивали тему перегрузки операций в предшествующих главах, а здесь предложим дополнительные детали и рассмотрим несколько широко применяемых методов перегрузки. Хотя мы не будем демонстрировать каждый из многочисленных доступных методов перегрузки операций, те методы, которые реализованы в главе, представляют собой достаточно большую репрезентативную выборку, чтобы исчерпывающе раскрыть возможности данной характеристики классов Python.

Основы

Вообще говоря, "перегрузка операций" просто означает *перехват* встроенных операций в методах класса — Python автоматически вызывает ваши методы, когда экземпляры класса обнаруживаются во встроенных операциях, и возвращаемое значение вашего метода становится результатом соответствующей операции. Ниже приведен обзор ключевых идей, лежащих в основе перегрузки.

- Перегрузка операций позволяет классам перехватывать нормальные операции Python.
- Классы могут перегружать все операции выражений Python.
- Классы также могут перегружать встроенные операции, такие как вывод, вызовы функций, доступ к атрибутам и т.д.
- Перегрузка делает экземпляры классов более похожими на встроенные типы.
- Перегрузка реализуется за счет предоставления особым образом именованных методов в классе.

Другими словами, когда в классе предоставляются особым образом именованные методы, тогда Python автоматически вызывает их в случае появления экземпляров данного класса в ассоциированных с ними выражениях. Ваш класс снабжает создаваемые из него объекты экземпляров поведением соответствующей операции.

Как вам уже известно, методы перегрузки операций не являются обязательными и обычно не имеют стандартных версий (кроме нескольких, которые ряд классов получают от `object`). Если вы не пишете код какого-то метода или не наследуете его, то это только означает, что ваш класс не поддерживает операцию, связанную с методом. Однако в случае использования такие методы позволяют классам эмулировать интерфейсы встроенных объектов и потому выглядеть более согласованными.

Конструкторы и выражения: `__init__` и `__sub__`

В качестве обзора рассмотрим следующий простой пример: класс `Number` из файла `number.py` предоставляет метод для перехвата создания экземпляра (`__init__`), а также метод для отлавливания выражений вычитания (`__sub__`). Специальные методы подобного рода являются привязками, которые дают возможность соединяться со встроенными операциями:

```
# Файл number.py

class Number:
    def __init__(self, start):                # Для Number(start)
        self.data = start
    def __sub__(self, other):                 # Для экземпляр - other
        return Number(self.data - other)      # Результатом будет новый экземпляр

>>> from number import Number                 # Извлечение класса из модуля
>>> X = Number(5)                             # Number.__init__(X, 5)
>>> Y = X - 2                                 # Number.__sub__(X, 2)
>>> Y.data                                    # Y является новым экземпляром Number
3
```

Ранее мы выяснили, что реализованный в коде метод конструктора `__init__` является наиболее употребительным методом перегрузки операций в Python; он присутствует в большинстве классов и применяется для инициализации вновь созданного объекта экземпляра с использованием любых аргументов, указываемых после имени класса. Метод `__sub__` исполняет роль бинарной операции аналогично методу `__add__` из введения главы 27, перехватывая выражения вычитания и возвращая в качестве своего результата новый экземпляр класса (попутно выполняя `__init__`).

Формально создание экземпляра сначала запускает метод `__new__`, который создает и возвращает новый объект экземпляра, передаваемый затем в метод `__init__` для инициализации. Тем не менее, поскольку метод `__new__` имеет встроенную реализацию и переопределяется лишь в крайне ограниченных ситуациях, почти все классы Python инициализируются за счет определения метода `__init__`. Один сценарий применения метода `__new__` будет показан при изучении *метаклассов* в главе 40; хотя и редко, но временами он также используется для настройки создания экземпляров неизменяемых типов.

Вы уже достаточно хорошо знаете метод `__init__` и методы базовых бинарных операций вроде `__sub__`, так что мы не будем здесь снова подробно исследовать их употребление. В главе мы раскроем ряд других инструментов, доступных в данной области, и предложим пример кода, который применяет их в распространенных сценариях использования.

Распространенные методы перегрузки операций

Почти все, что вы можете делать со встроенными объектами, такими как целые числа и списки, имеет соответствующие особым образом именованные методы для перегрузки в классах. Наиболее распространенные из них перечислены в табл. 30.1. В действительности многие методы перегрузки доступны в виде нескольких версий (например, `__add__`, `__radd__` и `__iadd__` для сложения), что является одной из причин настолько большого их количества. Полный список методов с особыми именами ищите в других книгах или в справочнике по языку Python.

Таблица 30.1. Наиболее распространенные методы перегрузки операций

Метод	Что реализует	Для чего вызывается
`__init__`	Конструктор	Создание объекта: `X = Class(args)`
`__del__`	Деструктор	Уничтожение объекта `X`
`__add__`	Операция +	`X + Y`, `X += Y`, если отсутствует `__iadd__`
`__or__`	Операция `|` (побитовое “ИЛИ”)	`X | Y`, `X |= Y`, если отсутствует `__ior__`
`__repr__`, `__str__`	Вывод, преобразования	`print(X)`, `repr(X)`, `str(X)`
`__call__`	Вызовы функций	`X(*args, **kargs)`
`__getattr__`	Извлечение атрибута	`X.undefined`
`__setattr__`	Присваивание атрибута	`X.any = value`
`__delattr__`	Удаление атрибута	`del X.any`
`__getattribute__`	Извлечение атрибута	`X.any`
`__getitem__`	Индексирование, нарезание, итерация	`X[key]`, `X[i:j]`, циклы `for` и другие итерационные конструкции, если отсутствует `__iter__`
`__setitem__`	Присваивание по индексу и срезу	`X[key] = value`, `X[i:j] = iterable`
`__delitem__`	Удаление по индексу и срезу	`del X[key]`, `del X[i:j]`
`__len__`	Длина	`len(X)`, проверки истинности, если отсутствует `__bool__`
`__bool__`	Булевские проверки	`bool(X)`, проверки истинности (в Python 2.X называется `__nonzero__`)
`__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`, `__ne__`	Сравнения	`X < Y`, `X > Y`, `X <= Y`, `X >= Y`, `X == Y`, `X != Y` (либо иначе `__cmp__` только в Python 2.X)
`__radd__`	Правосторонние операции	`Other + X`
`__iadd__`	Дополненные на месте операции	`X += Y` (либо иначе `__add__`)
`__iter__`, `__next__`	Итерационные контексты	`I=iter(X)`, `next(I)`; циклы `for`, `in`, если отсутствует `__contains__`, все включения, `map(F,X)`, остальные (`__next__` в Python 2.X называется `next`)
`__contains__`	Проверка членства	`item in X` (любой итерируемый объект)

Метод	Что реализует	Для чего вызывается
`__index__`	Целочисленное значение	`hex(X)`, `bin(X)`, `oct(X)`, `O[X]`, `O[X:]` (заменяет `__oct__`, `__hex__` из Python 2.X)
`__enter__`, `__exit__`	Диспетчер контекста (глава 34)	`with obj as var:`
`__get__`, `__set__`, `__delete__`	Атрибуты дескриптора (глава 38)	`X.attr`, `X.attr = value`, `del X.attr`
`__new__`	Создание (глава 40)	Создание объекта, перед `__init__`

Все методы перегрузки операций имеют имена, начинающиеся и заканчивающиеся двумя символами подчеркивания, чтобы отличать их от других имен, которые вы определяете в своих классах. Отображения специальных имен методов на выражения или операции предопределено языком Python и полностью документировано в стандартном руководстве по языку и других справочных ресурсах. Скажем, имя `__add__` всегда отображается на выражения + определением языка Python независимо от того, что фактически делает код метода `__add__`.

Хотя выражения запускают методы операций, остерегайтесь предположения о том, что существует преимущество в скорости, если исключить посредника и вызвать метод операции напрямую. На самом деле вызов метода операции напрямую может оказаться *в два раза медленнее*, вероятно из-за накладных расходов, связанных с вызовом функции, которых Python избегает или оптимизирует во встроенных операциях.

Ниже сравнивается скорость выполнения `len` и `__len__` с применением запускающего модуля Windows и методики измерения времени из главы 21 первого тома в Python 3.7 и 2.7: в обоих случаях вызов `__len__` напрямую занимает вдвое больше времени:

```
c:\code> py -3 -m timeit -n 1000 -r 5
             -s "L = list(range(100))" "x = L.__len__()"
1000 loops, best of 5: 0.134 usec per loop

c:\code> py -3 -m timeit -n 1000 -r 5
             -s "L = list(range(100))" "x = len(L)"
1000 loops, best of 5: 0.063 usec per loop

c:\code> py -2 -m timeit -n 1000 -r 5
             -s "L = list(range(100))" "x = L.__len__()"
1000 loops, best of 5: 0.117 usec per loop

c:\code> py -2 -m timeit -n 1000 -r 5
             -s "L = list(range(100))" "x = len(L)"
1000 loops, best of 5: 0.0596 usec per loop
```

Это не настолько неестественно, как может казаться — в одном известном научном учреждении мне действительно пришлось столкнуться с рекомендациями использовать более медленную альтернативу, чтобы достичь большей скорости!

Если методы перегрузки операций не определяются, тогда они могут быть унаследованы от суперклассов в точности как любые другие методы. Кроме того, все методы перегрузки операций необязательны – если вы не пишете код или не наследуете такой метод, то связанная с ним операция просто не поддерживается вашим классом, и попытка ее применения приводит к генерированию исключения. Некоторые встроенные операции наподобие вывода имеют стандартные реализации (наследуемые от подразумеваемого класса `object` в Python 3.X), но большая часть встроенных операций терпят неудачу для классов, если соответствующий метод перегрузки операции отсутствует.

Большинство методов перегрузки операций используются только в развитых программах, которые требуют, чтобы объекты вели себя аналогично встроенным объектам, хотя уже рассмотренный конструктор `__init__` присутствует в большей части классов. Давайте исследуем ряд дополнительных методов из табл. 30.1 на примерах.

Индексирование и нарезание: `__getitem__` и `__setitem__`

Наш первый набор методов позволяет классам имитировать некоторые линии поведения последовательностей и отображений. Если метод `__getitem__` определен в классе (или унаследован им), тогда он автоматически вызывается для операций индексирования экземпляров. В случае появления экземпляра X в выражении индексирования вроде `X[i]` интерпретатор Python вызывает метод `__getitem__`, унаследованный экземпляром, с передачей X в первом аргументе и индекса, указанного в квадратных скобках, во втором.

Например, следующий класс возвращает квадрат значения индекса – возможно нетипично, но иллюстративно для механизма в целом:

```
>>> class Indexer:
        def __getitem__(self, index):
            return index ** 2

>>> X = Indexer()
>>> X[2]                        # Для X[i] вызывается X.__getitem__(i)
4

>>> for i in range(5):
        print(X[i], end=' ')    # Каждый раз выполняется __getitem__(X, i)
0 1 4 9 16
```

Перехват срезов

Интересно отметить, что в дополнение к индексированию метод `__getitem__` также вызывается для *выражений срезов* – всегда в Python 3.X и условно в Python 2.X, если вы не предоставили более специфических методов нарезания. Говоря формально, встроенные типы обрабатывают нарезание тем же способом. Скажем, ниже демонстрируется нарезание для встроенного списка с применением верхней и нижней границ и страйда (см. главу 7):

```
>>> L = [5, 6, 7, 8, 9]
>>> L[2:4]                  # Нарезание с использованием синтаксиса: 2..(4-1)
[7, 8]
>>> L[1:]
[6, 7, 8, 9]
```

```
>>> L[:-1]
[5, 6, 7, 8]
>>> L[::2]
[5, 7, 9]
```

Однако на самом деле границы нарезания упаковываются в *объект среза* и передаются реализации индексирования списка. В действительности вы всегда можете передавать объект среза вручную — синтаксис среза по большей части является синтаксическим сахаром для индексирования объекта среза:

```
>>> L[slice(2, 4)]          # Нарезание с помощью объектов срезов
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, -1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

В классах с методом `__getitem__` это важно — в Python 3.X данный метод будет вызываться для базового индексирования (с индексом) и нарезания (с объектом среза). Наш предыдущий класс не обработает нарезание, потому что его логика предполагает передачу целочисленных индексов, но следующему классу такая обработка удастся. При вызове для *индексирования* аргументом является целое число, как и ранее:

```
>>> class Indexer:
        data = [5, 6, 7, 8, 9]
        def __getitem__(self, index): # Вызывается для индексирования или нарезания
            print('getitem:', index)
            return self.data[index] # Выполняется индексирование или нарезание

>>> X = Indexer()
>>> X[0]                   # Индексирование отправляет __getitem__ целое число
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9
```

Тем не менее, в случае вызова для *нарезания* метод принимает объект среза, который просто передается индексатору внедренного списка в новом выражении индексирования:

```
>>> X[2:4]                  # Нарезание отправляет __getitem__ объект среза
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

Там, где необходимо, метод `__getitem__` может проверять тип своего аргумента и извлекать границы объекта среза – объекты срезов имеют атрибуты `start`, `stop` и `step`, любой из которых можно опустить, указав `None`:

```
>>> class Indexer:
        def __getitem__(self, index):
            if isinstance(index, int):    # Проверка режима использования
                print('indexing', index)
            else:
                print('slicing', index.start, index.stop, index.step)

>>> X = Indexer()
>>> X[99]
indexing 99
>>> X[1:99:2]
slicing 1 99 2
>>> X[1:]
slicing 1 None None
```

Когда применяется метод присваивания по индексу `__setitem__`, он похожим образом перехватывает присваивания по индексу и срезу. Во втором случае в Python 3.X (и обычно в Python 2.X) он принимает объект среза, который может передаваться в другом присваивании по индексу либо использоваться напрямую таким же способом:

```
class IndexSetter:
   def __setitem__(self, index, value): # Перехватывает присваивание
                                        #    по индексу или срезу
      ...
      self.data[index] = value          # Присваивает по индексу или срезу
```

На самом деле метод `__getitem__` может автоматически вызываться даже в большем количестве контекстов, нежели индексация и нарезание – как вы вскоре увидите, он представляет собой также запасной вариант для *итерации*. Однако первым делом давайте взглянем на разновидность этих операций в Python 2.X и выясним потенциальную путаницу в данной категории.

Нарезание и индексирование в Python 2.X

В Python 2.X классы могут также определять методы `__getslice__` и `__setslice__` для перехвата извлечений и присваиваний по срезу специфическим образом. Если указанные методы переопределены, тогда им передаются границы выражения среза и для срезов с двумя пределами они предпочтительнее методов `__getitem__` и `__setitem__`. Во всех остальных случаях такой контекст работает так же, как в Python 3.X; скажем, объект среза по-прежнему создается и передается `__getitem__`, если метод `__getslice__` не найден или применяется расширенная форма среза с тремя пределами:

```
C:\code> c:\python27\python
>>> class Slicer:
        def __getitem__(self, index): print index
        def __getslice__(self, i, j): print i, j
        def __setslice__(self, i, j,seq): print i, j,seq

>>> Slicer()[1]       # Выполняется __getitem__ с int, как и в Python 3.X
1
>>> Slicer()[1:9] # Выполняется __getslice__, если присутствует, иначе __getitem__
1 9
>>> Slicer()[1:9:2]   # Выполняется __getitem__ с slice(), как и в Python 3.X!
slice(1, 9, 2)
```

В Python 3.X такие специфичные к срезам методы были *удалены*, поэтому даже в Python 2.X вы обычно должны взамен использовать `__getitem__` и `__setitem__` и допускать передачу в аргументах индексов и объектов срезов — ради прямой совместимости и во избежание необходимости обрабатывать срезы с двумя и тремя пределами по-разному. В большинстве классов все работает без какого-либо специального кода, потому что методам индексирования можно вручную передавать объект среза в квадратных скобках другого выражения индекса, как было показано в примере из предыдущего раздела. Еще один пример перехвата срезов приведен в разделе "Членство: `__contains__`, `__iter__` и `__getitem__`" далее в главе.

Но метод `__index__` в Python 3.X не имеет отношения к индексированию!

В качестве связанного замечания: при перехвате индексирования в Python 3.X не применяйте (вероятно, неудачно названный) метод `__index__` — он возвращает *целочисленное значение* для экземпляра и используется встроенными функциями, которые выполняют преобразование в строки цифр (оглядываясь назад, лучше бы его назвали, например, `__asindex__`):

```
>>> class C:
        def __index__(self):
            return 255

>>> X = C()
>>> hex(X)      # Целочисленное значение
'0xff'
>>> bin(X)
'0b11111111'
>>> oct(X)
'0o377'
```

Хотя данный метод не перехватывает индексирование экземпляров подобно `__getitem__`, он также применяется в контекстах, которые требуют целого числа — *включая* индексацию:

```
>>> ('C' * 256)[255]
'C'
>>> ('C' * 256)[X]      # Как индекс (не X[i])
'C'
>>> ('C' * 256)[X:]     # Как индекс (не X[i:])
'C'
```

В Python 2.X метод `__index__` работает точно так же, но не вызывается для встроенных функций `hex` и `oct`; для перехвата их вызовов в Python 2.X взамен используйте методы `__hex__` и `__oct__`.

Итерация по индексам: `__getitem__`

Существует привязка, которая не всегда очевидна для новичков, но оказывается удивительно полезной. При отсутствии более специфических методов итерации, которые будут представлены в следующем разделе, оператор `for` работает путем многократного индексирования последовательности от нуля до более высоких индексов, пока не обнаружится исключение выхода за границы `IndexError`. По этой причине `__getitem__` также оказывается одним из способов перегрузки итерации в Python —

если он определен, тогда циклы for на каждом проходе вызывают метод __getitem__ класса с последовательно увеличивающимися смещениями.

Мы имеем дело с ситуацией "реализовав одну возможность, получаем еще одну бесплатно" – любой встроенный или определяемый пользователем объект, который реагирует на индексацию, также реагирует на итерацию в цикле for:

```
>>> class StepperIndex:
        def __getitem__(self, i):
            return self.data[i]

>>> X = StepperIndex()          # X - объект StepperIndex
>>> X.data = "Spam"
>>>
>>> X[1]                        # Для индексирования вызывается __getitem__
'p'
>>> for item in X:              # Для циклов for вызывается __getitem__
        print(item, end=' ')    # Цикл for индексирует элементы 0..N
S p a m
```

На самом деле это случай "реализовав одну возможность, получаем бесплатно целый набор". Любой класс, поддерживающий циклы for, автоматически поддерживает все *итерационные контексты* в Python, многие из которых рассматривались в главах первого тома (итерационные контексты были представлены в главе 14 первого тома). Например, проверка членства in, списковые включения, встроенная функция map, присваивания списков и кортежей, а также конструкторы типов будут автоматически вызывать метод __getitem__, если он определен:

```
>>> 'p' in X                    # Для всех вызывается __getitem__
True

>>> [c for c in X]              # Списковое включение
['S', 'p', 'a', 'm']

>>> list(map(str.upper, X))     # Вызов map (в Python 3.X используйте list())
['S', 'P', 'A', 'M']

>>> (a, b, c, d) = X            # Присваивание последовательности
>>> a, c, d
('S', 'a', 'm')

>>> list(X), tuple(X), ''.join(X)    # И так далее...
(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')

>>> X
<__main__.StepperIndex object at 0x000000000297B630>
```

На практике такую методику можно применять для создания объектов, которые предоставляют интерфейс последовательностей, и для добавления логики, относящейся к операциям встроенных типов последовательностей; мы возвратимся к этой идее при расширении встроенных типов в главе 32.

Итерируемые объекты: __iter__ и __next__

Несмотря на то что описанный в предыдущем разделе подход с методом __getitem__ работает, в действительности он является просто запасным вариантом для итерации. В настоящее время все итерационные контексты языка Python перед __getitem__ будут сначала пытаться вызвать метод __iter__. То есть для многократного индексирования объекта они выбирают *протокол итерации*, который обсуждался

в главе 14 первого тома; попытка индексирования предпринимается, только если объект не поддерживает протокол итерации. Вообще говоря, вы также должны отдавать предпочтение методу `__iter__` — он лучше поддерживает общепринятые итерационные контексты, чем способен `__getitem__`.

Формально итерационные контексты работают путем передачи итерируемого объекта встроенной функции `iter` для вызова метода `__iter__`, который должен возвратить итерируемый объект. Когда этот метод `__next__` объекта итератора предоставлен, Python будет многократно вызывать его для выпуска элементов до тех пор, пока не сгенерируется исключение `StopIteration`. В качестве удобства для выполнения итерации вручную доступна также встроенная функция `next` — вызов `next(I)` представляет собой то же самое, что и `I.__next__()`. Сущность протокола итерации иллюстрировалась на рис. 14.1 в главе 14 первого тома.

Такой интерфейс итерируемых объектов имеет более высокий приоритет и опробуется первым. В случае если метод `__iter__` подобного рода не найден, тогда Python прибегает к схеме с `__getitem__` и как прежде многократно индексирует по смещениям, пока не возникнет исключение `IndexError`.

Примечание, касающееся нестыковки версий. Как упоминалось в главе 14 первого тома, только что описанный метод `I.__next__()` итератора в Python 2.X называется `I.next()`, а встроенная функция `next(I)` присутствует для совместимости — она вызывает `I.next()` в Python 2.X и `I.__next__()` в Python 3.X. Во всех остальных отношениях итераторы в Python 2.X работают аналогично.

Итерируемые объекты, определяемые пользователем

В схеме с методом `__iter__` классы реализуют определяемые пользователем итерируемые объекты, просто внедряя протокол итерации, представленный в главе 14 и детально исследованный в главе 20 первого тома. Например, в файле `squares.py` с показанным ниже содержимым используется класс для создания определяемого пользователем итерируемого объекта, который генерирует квадраты по запросу, а не выдает их все сразу (согласно предыдущей врезке "На заметку!" в Python 2.X необходимо применять `next` вместо `__next__` и `print` с финальной запятой):

```
# Файл squares.py

class Squares:
    def __init__(self, start, stop): # Сохранить состояние при создании
        self.value = start - 1
        self.stop = stop
    def __iter__(self):          # Получить объект итератора при вызове iter
        return self
    def __next__(self):          # Возвратить квадрат на каждой итерации
        if self.value == self.stop: # Также вызывается встроенной функцией next
            raise StopIteration
        self.value += 1
        return self.value ** 2
```

После импортирования его экземпляры могут появляться в итерационных контекстах в точности как встроенные объекты:

```
% python
>>> from squares import Squares
```

```
>>> for i in Squares(1, 5):    # for вызывает встроенную функцию iter,
                               #   которая вызывает __iter__
        print(i, end=' ')      # Каждая итерация вызывает __next__
1 4 9 16 25
```

Здесь объект итератора, возвращаемый `__iter__`, представляет собой просто экземпляр `self`, потому что метод `__next__` является частью самого класса `Squares`. В более сложных сценариях объект итератора может быть определен как отдельный класс и объект с собственной информацией о состоянии для поддержки множества активных итераций по тем же самым данным (вскоре мы рассмотрим пример). Об окончании итерации сообщается с помощью оператора `raise` языка Python — введенного в главе 29 и полностью раскрываемого в следующей части книги, но который всего лишь генерирует исключение, как если бы это сделал сам интерпретатор Python. Итерация вручную работает с определяемыми пользователем итерируемыми объектами так же, как и со встроенными типами:

```
>>> X = Squares(1, 5)       # Итерация вручную: то, что делают циклы
>>> I = iter(X)             # iter вызывает __iter__
>>> next(I)                 # next вызывает __next__  (в Python 3.X)
1
>>> next(I)
4
...остальные результаты не показаны...
>>> next(I)
25
>>> next(I)                 # Исключение можно перехватить в операторе try
StopIteration
```

Эквивалентная реализация такого итерируемого объекта посредством `__getitem__` может оказаться менее естественной, поскольку цикл `for` проходил бы тогда через все смещения с нуля и выше; передаваемые смещения были бы лишь косвенно связанными с диапазоном выпускаемых значений (`0..N` пришлось бы отображать на `start..stop`). Из-за того, что объекты `__iter__` предохраняют явно управляемое поведение между вызовами `next`, они могут быть более универсальными, чем `__getitem__`.

С другой стороны, итерируемые объекты, основанные на `__iter__`, временами могут быть более сложными и менее функциональными по сравнению с такими объектами, основанными на `__getitem__`. Они на самом деле предназначены для итерации, а не для произвольного индексирования — в них вообще не перегружается выражение индексирования, хотя их элементы можно собрать в последовательность вроде списка и сделать доступными другие операции:

```
>>> X = Squares(1, 5)
>>> X[1]
TypeError: 'Squares' object does not support indexing
Ошибка типа: объект Squares не поддерживает индексирование
>>> list(X)[1]
4
```

Единственный просмотр или множество просмотров

Схема с `__iter__` также реализована для всех остальных итерационных контекстов, которые мы видели в действии с методом `__getitem__` — проверка членства, конструкторы типов, присваивание последовательностей и т.д. Тем не менее, в отличие от предыдущего примера с `__getitem__` мы также должны знать, что метод

__iter__ класса может быть предназначен только для *единственного обхода*, а не для множества. Классы явным образом выбирают поведение просмотра в своем коде.

Скажем, поскольку текущий метод __iter__ класса Squares всегда возвращает self с только одной копией состояния итерации, он обеспечивает одноразовую итерацию; после итерации экземпляр данного класса становится пустым. Повторный вызов __iter__ на том же самом экземпляре снова возвращает self независимо от состояния, в котором он был оставлен. Как правило, для каждой новой итерации потребуется создавать новый итерируемый объект:

```
>>> X = Squares(1, 5)        # Создать итерируемый объект с состоянием
>>> [n for n in X]           # Израсходует элементы: __iter__ возвращает self
[1, 4, 9, 16, 25]
>>> [n for n in X]           # Теперь он пуст: __iter__ возвращает тот же self
[]

>>> [n for n in Squares(1, 5)]  # Создать новый итерируемый объект
[1, 4, 9, 16, 25]
>>> list(Squares(1, 3))      # Новый объект для каждого нового вызова __iter__
[1, 4, 9]
```

Для более прямой поддержки множества итераций мы могли бы также переписать код примера с использованием добавочного класса или другой методики, что мы вскоре и сделаем. Однако в том виде, как есть, за счет создания *нового экземпляра* для каждой итерации мы получаем свежую копию состояния итерации:

```
>>> 36 in Squares(1, 10)        # Другие итерационные контексты
True
>>> a, b, c = Squares(1, 3)     # Для каждого объекта вызывается __iter__
                                #   и затем __next__
>>> a, b, c
(1, 4, 9)
>>> ':'.join(map(str, Squares(1, 5)))
'1:4:9:16:25'
```

Подобно встроенным функциям с единственным просмотром, таким как map, преобразование в *список* тоже поддерживает множество просмотров, но ценой дополнительного расхода времени и пространства, которые могут быть, а могут и не быть важными в отдельно взятой программе:

```
>>> X = Squares(1, 5)
>>> tuple(X), tuple(X) # Второй вызов tuple() израсходует элементы в итераторе
((1, 4, 9, 16, 25), ())

>>> X = list(Squares(1, 5))
>>> tuple(X), tuple(X)
((1, 4, 9, 16, 25), (1, 4, 9, 16, 25))
```

После небольшого сравнения и противопоставления мы улучшим реализацию, чтобы более прямо поддерживать множество просмотров.

Классы или генераторы

Обратите внимание, что предыдущий пример, вероятно, мог бы стать проще, если бы в нем применялись *генераторные функции или выражения* — инструменты, представленные в главе 20 первого тома, которые автоматически выпускают итерируемые объекты и сохраняют состояние локальных переменных между итерациями:

```
>>> def gsquares(start, stop):
        for i in range(start, stop + 1):
            yield i ** 2

>>> for i in gsquares(1, 5):
        print(i, end=' ')

1 4 9 16 25

>>> for i in (x ** 2 for x in range(1, 6)):
        print(i, end=' ')

1 4 9 16 25
```

В отличие от классов генераторные функции и выражения неявно сохраняют свое состояние и создают методы, требуемые для соответствия протоколу итерации — с очевидными преимуществами в плане лаконичности кода в случае более простых примеров вроде показанных. С другой стороны, более явные атрибуты и методы класса, дополнительная структура, иерархии наследования и поддержка множества линий поведения могут лучше подходить в сложных сценариях использования.

Разумеется, в этом искусственном примере фактически можно было бы отказаться от обеих методик и просто применить цикл for, встроенную функцию map или списковое включение, чтобы построить сразу весь список. Тем не менее, если не учитывать данные о производительности, то самый лучший и быстрый способ решения задачи в Python часто также является самым простым:

```
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

Однако классы могут быть лучше при моделировании более сложных итераций, особенно когда они способны извлечь выгоду из средства классов вообще. Например, итерируемый объект, который выпускает элементы из сложного результата запроса к базе данных или веб-службе, может быть в состоянии полнее задействовать в своих интересах преимущество классов. В следующем разделе исследуется еще один сценарий использования для классов в итерируемых объектах, определяемых пользователем.

Множество итераторов в одном объекте

Ранее упоминалось о том, что объект итератора (с методом `__next__`), производимый итерируемым объектом, может быть определен как отдельный класс с собственной информацией о состоянии, чтобы более прямо поддерживать множество активных итераций по тем же самым данным. Посмотрим, что происходит, когда мы проходим по встроенному типу, подобному строке:

```
>>> S = 'ace'
>>> for x in S:
        for y in S:
            print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

Здесь внешний цикл захватывает итератор из строки, вызывая iter, и каждый вложенный цикл делает то же самое, чтобы получить независимый итератор. Поскольку каждый активный итератор имеет собственную информацию о состоянии, каждый цикл может поддерживать свою позицию в строке независимо от любых других активных циклов. Кроме того, мы не обязаны каждый раз создавать новую строку или преобразовывать в список; одиночный строковый объект сам по себе поддерживает множество просмотров.

Связанные примеры приводились в главах 14 и 20 первого тома. Скажем, генераторные функции и выражения, а также встроенные функции наподобие map и zip, оказались объектами одиночного итератора, соответственно поддерживающими единственный активный просмотр. В противоположность им встроенная функция range и другие встроенные типы вроде списков поддерживают множество активных итераторов с независимыми позициями.

При написании кода определяемых пользователем итерируемых объектов мы самостоятельно решаем, будут они поддерживать единственную активную итерацию или же много итераций. Чтобы достичь эффекта множества итераторов, методу __iter__ просто необходимо определить для итератора новый объект с состоянием взамен возвращения self в ответ на каждый запрос итератора.

Например, в приведенном далее коде класса SkipObject определяется итерируемый объект, который пропускает каждый второй элемент при выполнении итераций. Поскольку его объект итератора создается заново из дополнительного класса для каждой итерации, он поддерживает множество активных циклов напрямую (код находится в файле skipper.py):

```
#!python3
# Файл skipper.py

class SkipObject:
    def __init__(self, wrapped):    # Сохранить объект для использования
        self.wrapped = wrapped
    def __iter__(self):
        return SkipIterator(self.wrapped)     # Каждый раз новый итератор

class SkipIterator:
    def __init__(self, wrapped):
        self.wrapped = wrapped                # Информация о состоянии итератора
        self.offset = 0
    def __next__(self):
        if self.offset >= len(self.wrapped):  # Прекратить итерацию
            raise StopIteration
        else:
            item = self.wrapped[self.offset]  # Иначе возвратить и пропустить
            self.offset += 2
            return item

if __name__ == '__main__':
    alpha = 'abcdef'
    skipper = SkipObject(alpha)           # Создать объект контейнера
    I = iter(skipper)                     # Создать на нем итератор
    print(next(I), next(I), next(I))      # Посетить смещения 0, 2, 4

    for x in skipper:                     # for автоматически вызывает __iter__
        for y in skipper:                 # Вложенные for снова каждый раз
                                          #   вызывают __iter__
            print(x + y, end=' ')         # Каждый итератор имеет собственное
                                          #   состояние, смещение
```

Краткое замечание о переносимости: в том виде, как есть, это код Python 3.X. Чтобы сделать его совместимым с Python 2.X, импортируйте функцию print из Python 3.X и либо применяйте next вместо __next__ только в Python 2.X, либо определите псевдоним в области видимости класса для двойного использования в Python 2.X/3.X (как сделано в файле skipper_2x.py):

```
#!python
from __future__ import print_function   # Совместимость с Python 2.X/3.X
...
class SkipIterator:
    ...
    def __next__(self):
        ...
    next = __next__                       # Совместимость с Python 2.X/3.X
```

При запуске подходящей версии в одной из двух линеек Python пример работает подобно вложенным циклам со встроенными строками. Каждый активный цикл имеет собственную позицию в строке, потому что каждый получает независимый объект итератора, который записывает собственную информацию о состоянии:

```
% python skipper.py
a c e
aa ac ae ca cc ce ea ec ee
```

По контрасту с этим наш ранний пример `Squares` поддерживает только одну активную итерацию, если только мы не обращаемся к `Squares` снова во вложенных циклах для получения новых объектов. Здесь существует только один итерируемый объект `SkipObject` с множеством объектов итераторов, созданных из него.

Классы или срезы

Как и ранее, мы могли бы достичь похожих результатов с помощью встроенных инструментов – скажем, нарезания с третьей границей для пропуска элементов:

```
>>> S = 'abcdef'
>>> for x in S[::2]:
        for y in S[::2]:                  # Новые объекты на каждой итерации
            print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

Тем не менее, прием не совсем такой же по двум причинам. Во-первых, каждое выражение среза будет физически хранить весь результирующий список в памяти; с другой стороны, итерируемые объекты выпускают по одному значению за раз, что может сберечь солидное пространство в случае крупных результирующих списков.

Во-вторых, срезы производят *новые объекты*, поэтому в действительности мы не выполняли итерацию по тому же самому объекту в нескольких местах. Чтобы быть ближе к классу, нам пришлось бы создать единственный объект для прохода за счет заблаговременного нарезания:

```
>>> S = 'abcdef'
>>> S = S[::2]
>>> S
'ace'
>>> for x in S:
        for y in S:                       # Тот же самый объект, новые итераторы
            print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

Прием больше подобен решению на основе классов, но он по-прежнему хранит весь результат среза в памяти (в настоящее время не существует генераторной формы встроенного нарезания) и является единственным эквивалентом конкретного случая с пропуском каждого второго элемента.

Так как итерируемые объекты, определяемые пользователем, способны делать все то же, что и класс, они гораздо более универсальны, чем может вытекать из данного примера. Хотя такая универсальность требуется не во всех приложениях, определяемые пользователем итерируемые объекты представляют собой мощный инструмент — они позволяют заставить произвольные объекты выглядеть и вести себя подобно другим последовательностям и итерируемым объектам, с которыми мы сталкивались в книге. Скажем, мы могли бы применять такую методику с объектом базы данных для поддержки итерации по крупным выборкам из базы данных с множеством курсоров в одном и том же результате запроса.

Альтернативная реализация: `__iter__` плюс `yield`

А теперь обратимся к чему-то совершенно неявному, но потенциально полезному. В ряде приложений имеется возможность свести к минимуму требования к коду итерируемых объектов, определяемых пользователем, за счет *комбинирования* исследованного здесь метода `__iter__` и оператора генераторных функций `yield`, который обсуждался в главе 20. Поскольку генераторные функции *автоматически* сохраняют состояние локальных переменных и создают обязательные методы итераторов, они хорошо подходят для этой роли и дополняют предохранение состояния и другие полезные вещи, получаемые от классов.

Вспомните, что любая функция, которая содержит оператор `yield`, превращается в генераторную функцию. При вызове она возвращает новый *генераторный объект* с автоматическим предохранением локальной области видимости и позиции в коде, автоматически созданным методом `__iter__`, просто возвращающим сам объект, и автоматически созданным методом `__next__` (`next` в Python 2.X), запускающим функцию или возобновляющим ее выполнение с места, которое она оставила в последний раз:

```
>>> def gen(x):
        for i in range(x): yield i ** 2

>>> G = gen(5)    # Создание генераторного объекта с методами __iter__ и __next__
>>> G.__iter__() == G  # Оба метода существуют в том же самом объекте
True
>>> I = iter(G)        # Выполняется __iter__: генератор возвращает сам себя
>>> next(I), next(I)   # Выполняется __next__ (next в Python 2.X)
(0, 1)
>>> list(gen(5))    # Итерационные контексты автоматически выполняют iter и next
[0, 1, 4, 9, 16]
```

Все работает, даже если генераторная функция с оператором `yield` оказывается методом по имени `__iter__`: всякий раз, когда такой метод вызывается инструментом итерационного контекста, он будет возвращать новый генераторный объект с необходимым методом `__next__`. В качестве дополнительного бонуса генераторные функции, реализованные как методы в классах, имеют доступ к сохраненному состоянию в атрибутах экземпляров *и* в переменных локальной области видимости.

Например, следующий класс эквивалентен первоначальному определяемому пользователем итерируемому объекту `Squares`, код которого был написан ранее в файле `squares.py`:

```
# Файл squares_yield.py

class Squares:                          # Генератор на основе __iter__ + yield
```

```
    def __init__(self, start, stop):     # Метод __next__ является
                                         #    автоматическим/подразумеваемым
        self.start = start
        self.stop = stop
    def __iter__(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2
```

Здесь нет никакой необходимости создавать псевдоним next для __next__ ради совместимости с Python 2.X, потому что данный метод теперь является автоматическим и подразумеваемым посредством yield. Как и ранее, циклы for и другие итерационные инструменты автоматически проходят по экземплярам класса Squares:

```
% python
>>> from squares_yield import Squares
>>> for i in Squares(1, 5): print(i, end=' ')

1 4 9 16 25
```

И по обыкновению давайте посмотрим, как все это в действительности работает в итерационных контекстах. Прогон экземпляра класса Squares через iter позволяет получить результат вызова __iter__ обычным образом, но в нашем случае результатом будет генераторный объект с автоматически созданным методом __next__ того же самого рода, который мы всегда получаем при вызове генераторной функции, содержащей yield. Единственная разница здесь в том, что генераторная функция автоматически вызывается для iter. Обращение к интерфейсу next результирующего объекта производит результаты по запросу:

```
>>> S = Squares(1, 5)       # Выполняет метод __init__: класс сохраняет
                            #   состояние экземпляра
>>> S
<squares_yield.Squares object at 0x000000000294B630>
>>> I = iter(S)             # Выполняет метод __iter__: возвращает генератор
>>> I
<generator object __iter__ at 0x00000000029A8CF0>
>>> next(I)
1
>>> next(I)                 # Выполняет метод __next__ генератора
4
...и так далее...
>>> next(I)                 # Генератор содержит состояние экземпляров
                            #   и локальной области видимости
StopIteration
```

Также может помочь и тот факт, что мы могли бы назначить генераторному методу имя, отличающееся от __iter__, и вызывать его вручную для выполнения итерации – скажем, Squares(1,5).gen(). Использование имени __iter__, вызываемого автоматически итерационными инструментами, просто позволяет пропустить ручной шаг извлечения атрибута и вызова:

```
class Squares:              # Эквивалент с именем, отличающимся
                            #   от __iter__ (squares_manual.py)
    def __init__(...):
        ...
    def gen(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2
```

```
% python
>>> from squares_manual import Squares
>>> for i in Squares(1, 5).gen(): print(i, end=' ')
...те же самые результаты...

>>> S = Squares(1, 5)
>>> I = iter(S.gen())      # Вызвать генератор вручную для итерируемого
                           #   объекта/итератора
>>> next(I)
...те же самые результаты...
```

Реализация генератора как `__iter__` устраняет посредника в коде, хотя обе схемы в итоге создают новый генераторный объект для каждой итерации:

- при наличии `__iter__` итерация запускает метод `__iter__`, который возвращает новый генераторный объект с методом `__next__`;
- при отсутствии `__iter__` в коде производится вызов для создания генераторного объекта, метод `__iter__` которого возвращает сам объект.

Дополнительные сведения об операторе `yield` и генераторах ищите в главе 20 первого тома и сравните последнюю реализацию с более явной версией `__next__` из показанного ранее файла `squares.py`. Вы заметите, что новая версия `squares_yield.py` на 4 строки короче (7 вместо 11). В некотором смысле такая схема сокращает требования к коду классов почти так же, как функции замыканий из главы 17 первого тома, но в данном случае делает это с помощью комбинации методик функционального и объектно-ориентированного программирования, а не альтернативы в форме классов. Например, генераторный метод по-прежнему задействует атрибуты `self`.

Ряду наблюдателей решение может показаться содержащим слишком много уровней *магии* – оно опирается как на протокол итерации, так и на создание объектов генераторов, которые оба крайне неявные (в противоположность давнишним темам Python: см. `import this`). Помимо упомянутых мнений важно понимать также и не использующую `yield` разновидность итерируемых объектов на основе классов, поскольку она явная, универсальная и временами более широкая в смысле границ.

Однако методика с `__iter__`/`yield` может доказать свою эффективность в случаях, где она применима. Она также обеспечивает значительное преимущество, как объясняется в следующем разделе.

Множество итераторов с помощью `yield`

Кроме лаконичного кода итерируемый объект в виде определяемого пользователем класса из предыдущего раздела, основанный на комбинации `__iter__`/`yield`, предлагает важный дополнительный бонус – он также автоматически поддерживает *множество активных итераторов*. Это естественным образом следует из того факта, что каждое обращение к `__iter__` является вызовом генераторной функции, которая возвращает новый генераторный объект с собственной копией локальной области видимости для предохранения состояния:

```
% python
>>> from squares_yield import Squares   # Использование реализации Squares
                                        #   с __iter__/yield
>>> S = Squares(1, 5)
>>> I = iter(S)
>>> next(I); next(I)
1
4
```

```
>>> J = iter(S)  # Благодаря yield мы автоматически имеем множество итераторов
>>> next(J)
1
>>> next(I)      # I не зависит от J: собственное локальное состояние
9
```

Хотя генераторные функции являются итерируемыми объектами с единственным просмотром, неявные вызовы `__iter__` в итерационных контекстах обеспечивают поддержку новыми генераторами новых независимых просмотров:

```
>>> S = Squares(1, 3)
>>> for i in S:          # Каждый for вызывает __iter__
      for j in S:
         print('%s:%s' % (i, j), end=' ')

1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9
```

Реализация той же самой функциональности без `yield` требует добавочного класса, который будет явно и вручную сохранять состояние итератора, используя методики из предыдущего раздела (и код разрастается до 15 строк: на 8 больше, чем версия с `yield`):

```
# Файл squares_nonyield.py

class Squares:
   def __init__(self, start, stop):    # Генератор, не основанный на yield
      self.start = start       # Множество просмотров: дополнительный объект
      self.stop = stop
   def __iter__(self):
      return SquaresIter(self.start, self.stop)

class SquaresIter:
   def __init__(self, start, stop):
      self.value = start - 1
      self.stop = stop
   def __next__(self):
      if self.value == self.stop:
         raise StopIteration
      self.value += 1
      return self.value ** 2
```

Итоговый код работает аналогично версии на основе `yield` с множеством просмотров, но его объем больше и он более явный:

```
% python
>>> from squares_nonyield import Squares
>>> for i in Squares(1, 5): print(i, end=' ')

1 4 9 16 25
>>>
>>> S = Squares(1, 5)
>>> I = iter(S)
>>> next(I); next(I)
1
4
>>> J = iter(S)          # Множество итераторов без yield
>>> next(J)
1
>>> next(I)
9
```

```
>>> S = Squares(1, 3)
>>> for i in S:          # Каждый for вызывает __iter__
        for j in S:
            print('%s:%s' % (i, j), end=' ')

1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9
```

Наконец, подход на основе генераторов мог бы похожим образом устранить необходимость в добавочном классе итератора из предыдущего примера с пропуском элементов (`skipper.py`) благодаря его автоматическим методам и предохранению состояния локальных переменных (и получить 9 строк по сравнению с 16 в оригинале):

```
# Файл skipper_yield.py

class SkipObject:              # Еще один генератор на основе __iter__ + yield
    def __init__(self, wrapped):    # Область видимости экземпляра
                                    #   сохраняется нормально
        self.wrapped = wrapped      # Состояние локальной области видимости
                                    #   сохраняется автоматически
    def __iter__(self):
        offset = 0
        while offset < len(self.wrapped):
            item = self.wrapped[offset]
            offset += 2
            yield item
```

Результирующий код работает аналогично версии с множеством просмотров, не основанной на `yield`, но его объем меньше и он менее явный:

```
% python
>>> from skipper_yield import SkipObject
>>> skipper = SkipObject('abcdef')
>>> I = iter(skipper)
>>> next(I); next(I); next(I)
'a'
'c'
'e'
>>> for x in skipper:               # Каждый for вызывает __iter__:
                                    #   новый автоматический генератор
        for y in skipper:
            print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee
```

Конечно, все это искусственные примеры, которые можно было бы заменить более простыми инструментами вроде включений, и их код может как масштабироваться на реалистичные задачи, так и нет. Изучите предложенные альтернативы и сравните их. Как часто бывает в программировании, наилучший инструмент для работы других с высокой вероятностью окажется наилучшим инструментом и для вашей работы!

Членство: `__contains__`, `__iter__` и `__getitem__`

На самом деле история об итерациях даже обширнее, чем было показано до сих пор. Перегрузка операций часто *разделяется на уровни*: классы могут предоставлять специфические методы или более универсальные альтернативы, применяемые в качестве запасных вариантов. Ниже приведены примеры.

- Сравнения в Python 2.X используют специфические методы, такие как `__lt__` для "меньше, чем ", если он присутствует, либо иначе универсальный метод `__cmp__`. Как будет обсуждаться позже в главе, в Python 3.X применяются только специфические методы, но не `__cmp__`.
- Булевские проверки похожим образом сначала пробуют вызывать специфический метод `__bool__` (чтобы получить явный результат `True/False`) и при его отсутствии переходят на более универсальный метод `__len__` (ненулевая длина означает `True`). Как будет показано далее в главе, Python 2.X работает точно так же, но использует имя `__nonzero__` вместо `__bool__`.

В области итераций классы могут реализовывать операцию проверки членства `in` в виде итерации с применением методов `__iter__` или `__getitem__`. Тем не менее, для поддержки более специфической операции проверки членства в классах может быть предусмотрен метод `__contains__` – когда он присутствует, ему отдается предпочтение перед методом `__iter__`, который предпочтительнее `__getitem__`. Метод `__contains__` должен определять членство для *отображений* как применение к ключам (и может использовать быстрый просмотр), а для *последовательностей* как поиск.

Рассмотрим следующий класс, файл которого был снабжен возможностью для двойного применения в Python 2.X/3.X с использованием ранее описанных методик. Он реализует все три метода, а также тестирует проверку членства и разнообразные итерационные контексты, применяемые к экземпляру. При вызове его методы выводят трассировочные сообщения:

```
# Файл contains.py
from __future__ import print_function    # Совместимость с Python 2.X/3.X

class Iters:
    def __init__(self, value):
        self.data = value

    def __getitem__(self, i):            # Запасной вариант для итерации
        print('get[%s]:' % i, end='')    # Также для индексирования, нарезания
        return self.data[i]

    def __iter__(self):                  # Предпочтительнее для итерации
        print('iter=> ', end='')         # Допускает только один активный итератор
        self.ix = 0
        return self

    def __next__(self):
        print('next:', end='')
        if self.ix == len(self.data): raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item

    def __contains__(self, x):           # Предпочтительнее для операции in
        print('contains: ', end='')
        return x in self.data
    next = __next__                      # Совместимость с Python 2.X/3.X

if __name__ == '__main__':
    X = Iters([1, 2, 3, 4, 5])           # Создать экземпляр
    print(3 in X)                        # Членство
    for i in X:                          # Циклы for
        print(i, end=' | ')
```

```
print()
print([i ** 2 for i in X])        # Другие итерационные контексты
print( list(map(bin, X)) )

I = iter(X)                       # Итерация вручную
                                  #   (то, что делают другие контексты)
while True:
    try:
        print(next(I), end=' @ ')
    except StopIteration:
        break
```

В текущем виде класс из файла contains.py имеет метод __iter__, который поддерживает множество просмотров, но в любой момент времени активным может быть только один просмотр (например, вложенные циклы работать не будут), потому что каждая итерация пытается переустановить курсор просмотра в начало. Теперь, когда вам известно об операторе yield в методах итерации, вы должны быть в состоянии сказать, что показанный ниже код является эквивалентом, но разрешает множество активных просмотров. Сами судите, стоит ли его менее явная природа поддержки вложенных просмотров и сокращения объема на 6 строк (код находится в файле contains_yield.py):

```
class Iters:
    def __init__(self, value):
        self.data = value

    def __getitem__(self, i):            # Запасной вариант для итерации
        print('get[%s]:' % i, end='')    # Также для индексирования, нарезания
        return self.data[i]

    def __iter__(self):                  # Предпочтительнее для итерации
        print('iter=> next:', end='')    # Разрешает множество активных итераторов
        for x in self.data:              # __next__ для создания псевдонима
                                         #   next отсутствует
            yield x
            print('next:', end='')

    def __contains__(self, x):           # Предпочтительнее для операции in
        print('contains: ', end='')
        return x in self.data
```

Запуск любой из двух версий файла в Python 3.X и 2.X приводит к получению представленного далее вывода. Специфический метод __contains__ перехватывает операцию проверки членства, универсальный метод __iter__ перехватывает итерационные контексты, такие как вызываемый многократно метод __next__ (будь они записаны явно или подразумеваются оператором yield), а метод __getitem__ никогда не вызывается:

```
contains: True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100',
'0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

Однако посмотрите, что произойдет в выводе кода, если мы закомментируем метод __contains__ – теперь операция проверки членства направляется универсальному методу __iter__:

```
iter=> next:next:next:True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

Наконец, вот вывод в ситуации, когда закомментирован код методов `__contains__` и `__iter__` — для операции проверки членства и других итерационных контекстов вызывается запасной вариант индексирования `__getitem__` с последовательно растущими индексами, пока он не сгенерирует исключение `IndexError`:

```
get[0]:get[1]:get[2]:True
get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:['0b1', '0b10', '0b11', '0b100','0b101']
get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:
```

Как видно, метод `__getitem__` даже более универсален: помимо итераций он также перехватывает явное индексирование и нарезание. Выражения срезов запускают `__getitem__` с объектом среза, содержащим границы, как для встроенных типов, так и для определяемых пользователем классов, поэтому нарезание доступно в нашем классе автоматически:

```
>>> from contains import Iters
>>> X = Iters('spam')                # Индексирование
>>> X[0]                             # __getitem__(0)
get[0]:'s'

>>> 'spam'[1:]                       # Синтаксис нарезания
'pam'
>>> 'spam'[slice(1, None)]           # Объект среза
'pam'

>>> X[1:]                            # __getitem__(slice(..))
get[slice(1, None, None)]:'pam'
>>> X[:-1]
get[slice(None, -1, None)]:'spa'

>>> list(X)                          # И также итерация!
iter=> next:next:next:next:next:['s', 'p', 'a', 'm']
```

Но в более реалистичных сценариях использования итерации, не ориентированных на последовательности, метод `__iter__` может оказаться легче для реализации, т.к. он не обязан уметь обращаться с целочисленным индексом, а `__contains__` делает возможной оптимизацию поиска членства в качестве особого случая.

Доступ к атрибутам: `__getattr__` и `__setattr__`

Классы в Python также способны перехватывать базовый доступ к атрибутам (известный как уточнение), когда это необходимо или полезно. В частности, для *объекта*, созданного из класса, в коде может быть реализована операция точки *объект.атрибут*, участвующая в контекстах ссылки, присваивания и удаления. Ограниченный пример в рамках такой категории демонстрировался в главе 28, но здесь мы расширим данную тему.

Ссылка на атрибуты

Метод `__getattr__` перехватывает ссылки на атрибуты. Он вызывается с именем атрибута в виде строки всякий раз, когда вы пытаетесь уточнить экземпляр с помощью *неопределенного* (несуществующего) имени атрибута. Метод `__getattr__` *не* вызывается, если Python может найти атрибут с применением процедуры поиска в дереве наследования.

Из-за своего поведения метод `__getattr__` удобен в качестве привязки, обеспечивающей реагирование на запросы атрибутов в обобщенной манере. Он обычно используется для делегирования вызовов внедренным (или "вложенным") объектам из промежуточного объекта контроллера — подобно тому, как было представлено во введении в *делегирование* в главе 28. Метод `__getattr__` также может применяться для адаптации классов к интерфейсу или последующего добавления *средств доступа* к атрибутам — логики в методе, которая проверяет достоверность или вычисляет значение атрибута после того, как он уже используется через простую точечную запись.

Базовый механизм, лежащий в основе достижения указанных целей, прямолинеен. В приведенном ниже классе перехватываются ссылки на атрибуты, динамически вычисляется значение для одного атрибута и генерируется ошибка для остальных неподдерживаемых атрибутов с помощью оператора `raise`, описанного ранее в главе при рассмотрении итераторов (и полностью раскрываемого в части VII):

```
>>> class Empty:
        def __getattr__(self, attrname): # Вызывается для неопределенного
                                         #  атрибута в self
            if attrname == 'age':
                return 40
            else:
                raise AttributeError(attrname)

>>> X = Empty()
>>> X.age
40
>>> X.name
...текст сообщения об ошибке не показан...
AttributeError: name
Ошибка атрибута: name
```

Здесь класс `Empty` и его экземпляр `X` не имеют собственных реальных атрибутов, поэтому доступ к `X.age` направляется методу `__getattr__`; `self` присваивается экземпляр (`X`), а `attrname` — строка с именем неопределенного атрибута (`'age'`). Класс делает `age` похожим на реальный атрибут, возвращая действительное значение в качестве результата выражения уточнения `X.age` (`40`). В сущности, `age` становится *динамически вычисляемым* атрибутом — его значение формируется выполняющимся кодом, а не извлечением какого-то объекта.

Для атрибутов, которые класс не знает, как обрабатывать, метод `__getattr__` генерирует встроенное исключение `AttributeError`, сообщая Python о том, что они являются по-настоящему неопределенными именами; обращение к `X.name` инициирует ошибку. Вы увидите метод `__getattr__` в действии снова, когда мы займемся делегированием и свойствами в последующих двух главах; а пока давайте перейдем к исследованию связанных инструментов.

Присваивание и удаление атрибутов

В рамках той же области метод `__setattr__` перехватывает *все* присваивания значений атрибутам. Если данный метод определен или унаследован, тогда выражение `self.`*атрибут* `=` *значение* становится `self.__setattr__('`*атрибут*`',` *значение*`)`. Подобно `__getattr__` это позволяет классу перехватывать изменения атрибутов и выполнять желаемые проверки достоверности или преобразования.

Тем не менее, использовать метод `__setattr__` несколько сложнее, т.к. присваивание значения любому атрибуту в `self` внутри `__setattr__` снова вызывает `__setattr__`, что потенциально может стать причиной бесконечного *цикла рекурсии* (и довольно быстро привести к исключению, связанному с переполнением стека!). На самом деле сказанное применимо ко всем присваиваниям атрибутов в `self` где угодно в классе — все они направляются методу `__setattr__`, даже присваивания, находящиеся в других методах, и присваивания именам, отличным от тех, которые могут запускать метод `__setattr__` в первую очередь. Не забывайте, что метод `__setattr__` перехватывает *все* присваивания значений атрибутам.

Если вы хотите использовать метод `__setattr__`, то можете избежать циклов, реализуя присваивания значений атрибутам экземпляра в виде присваиваний ключам словаря атрибутов. То есть применяйте `self.__dict__['name'] = x`, но не `self.name = x`; из-за того, что вы не выполняете присваивание самому `__dict__`, цикл не возникает:

```
>>> class Accesscontrol:
        def __setattr__(self, attr, value):
            if attr == 'age':
                self.__dict__[attr] = value + 10     # Не self.имя = значение
                                                     #   или setattr
            else:
                raise AttributeError(attr + ' not allowed')  # не разрешено

>>> X = Accesscontrol()
>>> X.age = 40                                       # Вызывается __setattr__
>>> X.age
50
>>> X.name = 'Bob'
...текст не показан...
AttributeError: name not allowed
Ошибка атрибута: name не разрешено
```

Если вы замените присваивание `__dict__` любым из следующих присваиваний, тогда возникнет бесконечный цикл рекурсии и затем исключение — точечная запись и ее эквивалент в виде встроенной функции `setattr` (аналог `getattr`, отвечающий за присваивание) потерпят неудачу в случае присваивания `age` за пределами класса:

```
self.age = value + 10                # Зацикливание
setattr(self, attr, value + 10)      # Зацикливание (attr является 'age')
```

Присваивание значения другому имени внутри класса тоже инициирует рекурсивный вызов метода `__setattr__`, хотя в данном классе результатом будет менее драматичное ручное исключение `AttributeError`:

```
self.other = 99     # Рекурсивный вызов, но без зацикливания: терпит неудачу
```

Избежать рекурсивного зацикливания в классе, где используется `__setattr__`, возможно также путем направления любых присваиваний значений атрибутам распо-

ложенному выше суперклассу посредством вызова, а не присваивания значений ключам в `__dict__`:

```
self.__dict__[attr] = value + 10                # Нормально: зацикливания нет
object.__setattr__(self, attr, value + 10)  # Нормально: зацикливания нет
                                            #   (только классы нового стиля)
```

Однако поскольку форма `object` требует применения классов нового стиля в Python 2.X, мы отложим рассмотрение деталей до главы 38, где исследуется управление атрибутами в целом.

Третий метод управления атрибутами, `__delattr__`, принимает строку с именем атрибута и вызывается для всех удалений атрибутов (т.е. `del` *`объект.атрибут`*). Как и `__setattr__`, он должен избегать рекурсивного зацикливания, направляя удаления атрибутов с использованием класса через `__dict__` или суперкласс.

В главе 32 мы узнаем, что атрибуты, реализованные с помощью средств классов нового стиля, таких как *слоты* и *свойства*, физически не хранятся в словаре пространства имен `__dict__` экземпляра (и слоты могут даже препятствовать его существованию!). По указанной причине код, где желательно поддерживать такие атрибуты, для присваивания им значений должен применять `__setattr__` с показанной здесь схемой `object.__setattr__`, а не индексирование `self.__dict__`, которое используется только когда известно, что участвующие классы хранят все свои данные в самом экземпляре. В главе 38 мы также увидим, что метод нового стиля `__getattribute__` имеет похожие требования. Такое изменение обязательно в Python 3.X, но также применяется к Python 2.X, если используются классы нового стиля.

Другие инструменты управления атрибутами

Рассмотренные выше методы для перегрузки операций доступа к атрибутам позволяют управлять или специализировать доступ к атрибутам в объектах. Как правило, они исполняют узкоспециализированные роли, часть которых мы исследуем позже в книге. За еще одним примером метода `__getattr__` в работе обращайтесь в файл `person-composite.py` из главы 28. Кроме того, имейте в виду, что в Python существуют и другие способы управления доступом к атрибутам.

- Метод `__getattribute__` перехватывает операции извлечения всех атрибутов, не только тех, которые не определены, но при его применении вы должны соблюдать большую осторожность, чем с `__getattr__`, чтобы избегать зацикливания.
- Встроенная функция `property` позволяет ассоциировать методы с операциями извлечения и установки для специфического атрибута класса.
- Дескрипторы предоставляют протокол для ассоциирования методов `__get__` и `__set__` класса с операциями доступа к специфическому атрибуту класса.
- Атрибуты слотов объявляются в классах, но создают неявное хранилище в каждом экземпляре.

Поскольку перечисленные инструменты относительно сложны и интересны далеко не каждому программисту на Python, мы отложим рассмотрение свойств до главы 32, а всех методик управления атрибутами — до главы 38.

Эмуляция защиты атрибутов экземпляра: часть 1

В следующем коде (`private0.py`) демонстрируется другой сценарий использования таких инструментов. Он обобщает предыдущий пример, чтобы позволить каждому подклассу иметь собственный список закрытых имен, которым экземпляры не могут присваивать значения (и применяет определяемый пользователем класс исключения, что обсуждается в части VII):

```
class PrivateExc(Exception): pass              # Исключения подробно
                                               #   рассматриваются в части VII

class Privacy:
    def __setattr__(self, attrname, value):    # Вызывается для
                                               #   self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self)   # Сгенерировать определяемое
                                               #   пользователем исключение
        else:
            self.__dict__[attrname] = value    # Избежать зацикливания,
                                               #   используя ключ словаря

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Tom'          # Чтобы сделать лучше,
                                               #   обратитесь в главу 39!

if __name__ == '__main__':
    x = Test1()
    y = Test2()

    x.name = 'Bob'                             # Работает
    #y.name = 'Sue'                            # Терпит неудачу
    print(x.name)

    y.age = 30                                 # Работает
    #x.age = 40                                # Терпит неудачу
    print(y.age)
```

Фактически это первое пробное решение для реализации *защиты атрибутов* в Python — запрет вносить изменения в имена атрибутов за пределами класса. Хотя Python не поддерживает закрытые объявления как таковые, методики вроде показанной здесь способны эмулировать большую часть их предназначения.

Тем не менее, решение получилось неполное и даже неуклюжее. Чтобы сделать его более эффективным, мы должны дополнить классы возможностью устанавливать свои закрытые атрибуты более естественным путем, не проходя каждый раз через `__dict__`, как обязан поступать конструктор во избежание запуска метода `__setattr__` и генерации исключения. Лучший и более совершенный подход может требовать класса-оболочки ("посредника") для препятствования операциям доступа к закрытым атрибутам, выполняемым только за пределами класса, а также метода `__getattr__` для проверки операций извлечения атрибутов.

Мы отложим полное решение по защите атрибутов до главы 39, где будем использовать *декораторы классов* для более общего перехвата и проверки атрибутов. Однако, несмотря на то, что защиту атрибутов можно эмулировать подобным образом, на практике так почти никогда не поступают. Программисты на Python в состоянии

разрабатывать крупные объектно-ориентированные фреймворки и приложения без закрытых объявлений — интересные сведения о контроле доступа в целом, которые выходят за рамки преследуемых здесь целей.

Тем не менее, перехват ссылок и присваиваний атрибутов в общем случае является полезным приемом; он поддерживает *делегирование* — методику проектирования, которая позволяет управляющим объектам становиться оболочками для внедренных объектов, добавлять новые линии поведения и направлять другие операции на выполнение внедренным объектам. Из-за того, что делегирование и внедренные объекты задействуют темы, связанные с проектированием, мы возвратимся к ним в следующей главе.

Строковое представление: `__repr__` и `__str__`

Рассматриваемые в настоящем разделе методы имеют дело с форматами отображения — тема, которую мы уже исследовали в предшествующих главах, но подытожим и формализуем здесь. В приведенном ниже коде используются конструктор `__init__` и метод перегрузки `__add__`, с которыми мы встречались ранее (+ представляет собой операцию на месте, просто чтобы показать, что она может тут присутствовать; согласно главе 27 именованный метод возможно предпочтительнее). Как нам известно, стандартное отображение объектов экземпляров для класса вроде этого не приносит особой пользы и не может считаться эстетически привлекательным:

```
>>> class adder:
        def __init__(self, value=0):
            self.data = value           # Инициализировать данные
        def __add__(self, other):
            self.data += other          # Добавить other на месте (плохая форма?)

>>> x = adder()                         # Стандартные отображения
>>> print(x)
<__main__.adder object at 0x00000000029736D8>
>>> x
<__main__.adder object at 0x00000000029736D8>
```

Но реализация или наследование методов строкового представления позволяет настраивать отображение — как показано в следующем примере, где в подклассе определяется метод `__repr__`, который возвращает строковое представление для своих экземпляров:

```
>>> class addrepr(adder):               # Унаследовать __init__, __add__
        def __repr__(self):             # Добавить строковое представление
            return 'addrepr(%s)' % self.data # Преобразовать в строку как в коде

>>> x = addrepr(2)                      # Выполняется __init__
>>> x + 1                               # Выполняется __add__ (x.add() лучше?)
>>> x                                   # Выполняется __repr__
addrepr(3)
>>> print(x)                            # Выполняется __repr__
addrepr(3)
>>> str(x), repr(x)                     # Выполняется __repr__ для обоих
('addrepr(3)', 'addrepr(3)')
```

В случае определения метода `__repr__` (или близкородственного ему `__str__`) он автоматически вызывается, когда экземпляры класса выводятся или преобразуются в строки. Упомянутые методы позволяют определять улучшенный формат отоб-

ражения для объектов по сравнению со стандартным отображением экземпляров. В методе `__repr__` с помощью базового форматирования строк управляемый объект `self.data` преобразуется в более дружественную к человеку строку, предназначенную для отображения.

Для чего используются два метода отображения?

Все увиденное до сих пор по большому счету было обзором. Но наряду с тем, что эти методы обычно просты в применении, их роли и поведение имеют тонкие последствия как для проектирования, так и для написания кода. В частности, Python предлагает два метода отображения, призванные поддерживать отличающееся отображение для разной аудитории.

- Метод `__str__` сначала опробуется для операции `print` и встроенной функции `str` (внутренний эквивалент которой запускает операция `print`). В общем случае он должен возвратить отображение, дружественное к пользователю.
- Метод `__repr__` используется во всех остальных контекстах: для эхо-вывода в интерактивной подсказке, функции `repr` и вложенных появлений, а также `print` и `str`, если метод `__str__` отсутствует. В общем случае он должен возвратить строку как в коде, которую можно было бы применять для воссоздания объекта, или детальное отображение для разработчиков.

То есть `__repr__` используется везде, исключая `print` и `str`, когда метод `__str__` определен. Это означает, что вы можете реализовать метод `__repr__` для определения единственного формата отображения, применяемого повсюду, и метод `__str__` либо для поддержки единственно `print` и `str`, либо чтобы предоставить для них альтернативное отображение.

Как отмечалось в главе 28, универсальные инструменты могут также отдавать предпочтение использованию `__str__` и оставлять другим классам возможность добавления альтернативного отображения `__repr__` для применения в остальных контекстах до тех пор, пока инструменту достаточно отображений `print` и `str`. И наоборот, универсальный инструмент, который реализует `__repr__`, по-прежнему оставляет клиентам возможность добавления альтернативного отображения с помощью `__str__` для `print` и `str`. Другими словами, если реализуется один из двух методов, то другой будет доступным для дополнительного отображения. В случаях, где выбор неочевиден, метод `__str__` обычно предпочтительнее использовать для более крупных отображений, дружественных к пользователю, а `__repr__` – для низкоуровневых или отображений как в коде и включающих все ролей.

Давайте напишем код, более конкретно иллюстрирующий отличия между двумя методами. В предыдущем примере из этого раздела было показано, как `__repr__` применяется в качестве запасного варианта во многих контекстах. Однако хотя вывод прибегает к `__repr__`, если метод `__str__` не определен, противоположное неверно – другие контексты, такие как эхо-вывод в интерактивной подсказке, используют только `__repr__` и вообще не пытаются обращаться к `__str__`:

```
>>> class addstr(adder):
        def __str__(self):                        # __str__, но не __repr__
            return '[Value: %s]' % self.data  # Преобразовать в симпатичную строку

>>> x = addstr(3)
>>> x + 1
>>> x                                             # По умолчанию __repr__
<__main__.addstr object at 0x00000000029738D0>
```

```
>>> print(x)                                  # Выполняется __str__
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', '<__main__.addstr object at 0x00000000029738D0>')
```

По этой причине метод `__repr__` может оказаться лучше, если вы хотите иметь *единственное* отображение для всех контекстов. Тем не менее, за счет определения обоих методов вы можете поддерживать в разных контекстах отличающиеся отображения – например, отображение посредством `__str__` для конечного пользователя и низкоуровневое отображение с помощью `__repr__` для применения программистами во время разработки. В действительности `__str__` просто переопределяет `__repr__` для контекстов отображения, более дружественных к пользователю:

```
>>> class addboth(adder):
        def __str__(self):
            return '[Value: %s]' % self.data  # Строка, дружественная
                                              #   к пользователю
        def __repr__(self):
            return 'addboth(%s)' % self.data  # Строка как в коде

>>> x = addboth(4)
>>> x + 1
>>> x                                         # Выполняется __repr__
addboth(5)
>>> print(x)                                  # Выполняется __str__
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

Замечания по использованию отображения

Несмотря на простоту использования в целом, я должен привести три замечания относительно этих методов. Во-первых, имейте в виду, что `__str__` и `__repr__` обязаны возвращать *строки*; другие результирующие типы не преобразуются и вызывают ошибки, так что при необходимости обеспечьте их обработку инструментом преобразования в строку (скажем, `str` или `%`).

Во-вторых, в зависимости от логики преобразования в строки дружественное к пользователю отображение `__str__` может применяться, только когда объекты находятся на верхнем уровне операции `print`; объекты, *вложенные* внутрь более крупных объектов, могут по-прежнему выводиться посредством `__repr__` либо их стандартных методов. Оба аспекта иллюстрируются в следующем взаимодействии:

```
>>> class Printer:
        def __init__(self, val):
            self.val = val
        def __str__(self):             # Используется для самого экземпляра
            return str(self.val)       # Преобразовать в строковый результат

>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)          # __str__ выполняется при выводе экземпляра
                             # Но не в ситуации, когда экземпляр находится в списке!
2
3
>>> print(objs)
[<__main__.Printer object at 0x00000000297AB38>, <__main__.Printer obj...
etc...>]
```

```
>>> objs
[<__main__.Printer object at 0x000000000297AB38>, <__main__.Printer obj...
etc...>]
```

Чтобы обеспечить использование специального отображения во всех контекстах независимо от контейнера, необходимо реализовывать метод `__repr__`, а не `__str__`; первый из них выполняется во всех случаях, если второй неприменим, включая вложенные появления:

```
>>> class Printer:
        def __init__(self, val):
            self.val = val
        def __repr__(self):     # __repr__ используется print,
                                #  если отсутствует __str__
          return str(self.val) # __repr__ используется при эхо-выводе
                                #  в интерактивной подсказке
                                #  или при вложенном появлении
>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)    # __str__ отсутствует: выполняется __repr__
2
3
>>> print(objs)                # Выполняется __repr__, не __str__
[2, 3]
>>> objs
[2, 3]
```

В-третьих, и вероятно это самый тонкий момент, в редких контекстах методы отображения также потенциально могут приводить к бесконечным *циклам рекурсии* — поскольку отображение некоторых объектов предусматривает отображение других объектов, не исключено, что какое-то отображение может запустить отображение выводимого объекта, образуя цикл. Ситуация достаточно редкая и малоизвестная, чтобы не затрагивать ее здесь, но потенциальная возможность зацикливания `__repr__` обсуждается во врезке "На заметку!" после кода класса `ListInherited` в следующей главе.

На практике метод `__str__` и более охватывающий родственный ему метод `__repr__`, похоже, являются вторыми по частоте использования методами перегрузки операций в Python после `__init__`. В любое время, когда вы в состоянии выводить объект и видеть его специальное отображение, вероятно, задействован один из этих двух инструментов. В главах 28 и 31 можно найти дополнительные примеры применения данных инструментов и связанные с ними проектные компромиссы, а в главе 35 описана их роль в классах исключений, где метод `__str__` более востребован, чем `__repr__`.

Использование с правой стороны и на месте: `__radd__` и `__iadd__`

Следующая группа методов перегрузки расширяет функциональность методов бинарных операций, таких как `__add__` и `__sub__` (вызываемых для + и -), которые мы уже видели. Как упоминалось ранее, одна из причин существования настолько большого количества методов перегрузки операций связана с тем, что они встречаются во многих разновидностях — для каждого бинарного выражения мы можем реализовать варианты *с левой стороны*, *с правой стороны* и *на месте*. Хотя также применяются стандартные реализации, когда не написан код для всех трех вариантов, именно роли ваших объектов диктуют, код скольких вариантов потребуется предоставить.

Правостороннее сложение

Например, реализованные до настоящего времени методы __add__ формально не поддерживают использование объектов экземпляров с правой стороны операции +:

```
>>> class Adder:
        def __init__(self, value=0):
            self.data = value
        def __add__(self, other):
            return self.data + other

>>> x = Adder(5)
>>> x + 2
7
>>> 2 + x
TypeError: unsupported operand type(s) for +: 'int' and 'Adder'
Ошибка типа: неподдерживаемый тип (типы) операнда для +: int и Adder
```

Чтобы реализовать более универсальные выражения и тем самым поддерживать коммутативные операции, необходимо также написать код метода __radd__. Интерпретатор Python вызывает __radd__, только когда объектом с правой стороны операции + является экземпляр вашего класса, но объект с левой стороны не относится к экземплярам вашего класса. Во всех остальных случаях для объекта с левой стороны взамен вызывается метод __add__ (представленные в этом разделе пять классов, Commuter1 — Commuter5, вместе с кодом самотестирования находятся в файле commuter.py):

```
class Commuter1:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    def __radd__(self, other):
        print('radd', self.val, other)
        return other + self.val

>>> from commuter import Commuter1
>>> x = Commuter1(88)
>>> y = Commuter1(99)

>>> x + 1                # __add__: экземпляр + не экземпляр
add 88 1
89
>>> 1 + y                # __radd__: не экземпляр + экземпляр
radd 99 1
100
>>> x + y                # __add__: экземпляр + экземпляр, запускает __radd__
add 88 <commuter.Commuter1 object at 0x00000000029B39E8>
radd 99 88
187
```

Обратите внимание на реверсирование порядка в __radd__: на самом деле self находится с правой стороны операции +, а other — с левой стороны. Также учтите, что x и y здесь являются экземплярами того же самого класса; когда в выражении смешиваются экземпляры разных классов, Python отдает предпочтение классу экземпляра с левой стороны. При сложении двух экземпляров Python выполняет метод __add__, который в свою очередь запускает __radd__, упрощая левый операнд.

Повторное использование __add__ в __radd__

Для подлинно коммутативных операций, которые не требуют учета позиции, иногда вполне достаточно повторно использовать __add__ для __radd__: вызвать __add__ напрямую; поменять местами операнды и снова выполнить сложение, чтобы запустить __add__ косвенно; либо просто назначить __radd__ в качестве псевдонима для __add__ на верхнем уровне оператора class (т.е. в области видимости класса). Показанные далее альтернативные версии реализуют три упомянутые схемы и возвращают те же результаты, что и первоначальная версия – хотя последняя экономит один вызов или направление и потому может оказаться быстрее (во всех версиях __radd__ вызывается, когда self находится с правой стороны операции +):

```
class Commuter2:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    def __radd__(self, other):
        return self.__add__(other)        # Явно вызвать __add__

class Commuter3:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    def __radd__(self, other):
        return self + other               # Поменять местами и снова сложить

class Commuter4:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    __radd__ = __add__                    # Псевдоним: исключить посредника
```

Во всех версиях правосторонние появления экземпляров приводят к запуску единственного разделяемого метода __add__ с передачей для self правого операнда, чтобы трактовать его точно так же, как левостороннее появление. Для лучшего понимания реализации запустите все версии самостоятельно; возвращаемые ими значения будут такими же, как у исходной версии.

Распространение типа класса

В более реалистичных классах, где может возникать необходимость в распространении типа класса на результаты, ситуация становится сложнее: чтобы выяснить, безопасно ли выполнять преобразование, и таким образом избежать вложенности, иногда требуется проверка типа. Скажем, в следующем коде без проверки isinstance мы могли бы в итоге получить экземпляр Commuter5, атрибутом val которого оказался бы еще один экземпляр Commuter5, когда выполняется сложение двух экземпляров и __add__ запускает __radd__:

```
class Commuter5:                      # Распространение типа класса на результаты
    def __init__(self, val):
        self.val = val
```

```
    def __add__(self, other):
        if isinstance(other, Commuter5):       # Проверка типа во избежание
                                               #   вложенности объектов
            other = other.val
        return Commuter5(self.val + other)     # Иначе результатом операции +
                                               #   является еще один Commuter5
    def __radd__(self, other):
        return Commuter5(other + self.val)
    def __str__(self):
        return '<Commuter5: %s>' % self.val
```

```
>>> from commuter import Commuter5
>>> x = Commuter5(88)
>>> y = Commuter5(99)
>>> print(x + 10)          # Результат - еще один экземпляр Commuter5
<Commuter5: 98>
>>> print(10 + y)
<Commuter5: 109>

>>> z = x + y              # Не вложенный: не вызывает рекурсивно __radd__
>>> print(z)
<Commuter5: 187>
>>> print(z + 10)
<Commuter5: 197>
>>> print(z + z)
<Commuter5: 374>
>>> print(z + z + 1)
<Commuter5: 375>
```

Необходимость проверки типа с помощью `isinstance` здесь очень тонкая — закомментируйте проверку, запустите и отследите, чтобы понять, почему она обязательна. Вы заметите, что в последней части предыдущего теста получаются отличающиеся и вложенные объекты, которые по-прежнему корректно выполняют математические действия, но инициируют бессмысленные рекурсивные вызовы для упрощения своих значений, а добавочные вызовы конструктора формируют результаты:

```
>>> z = x + y              # Проверка с помощью isinstance закомментирована
>>> print(z)
<Commuter5: <Commuter5: 187>>
>>> print(z + 10)
<Commuter5: <Commuter5: 197>>
>>> print(z + z)
<Commuter5: <Commuter5: <Commuter5: <Commuter5: 374>>>>
>>> print(z + z + 1)
<Commuter5: <Commuter5: <Commuter5: <Commuter5: 375>>>>
```

Для тестирования остаток содержимого файла `commuter.py` должен выглядеть и выполняться так, как показано ниже — классы вполне естественно могут появляться в кортежах:

```
#!python
from __future__ import print_function    # Совместимость с Python 2.X/3.X
...здесь определены классы...

if __name__ == '__main__':
    for klass in (Commuter1, Commuter2, Commuter3, Commuter4, Commuter5):
        print('-' * 60)
        x = klass(88)
```

```
        y = klass(99)
        print(x + 1)
        print(1 + y)
        print(x + y)
c:\code> commuter.py
-------------------------------------------------------------
add 88 1
89
radd 99 1
100
add 88 <__main__.Commuter1 object at 0x000000000297F2B0>
radd 99 88
187
-------------------------------------------------------------
...и так далее...
```

Вариантов реализации слишком много, чтобы их можно было все здесь исследовать, а потому для лучшего понимания поэкспериментируйте с предложенными классами самостоятельно; например, назначение псевдонима `__radd__` методу `__add__` в `Commuter5` не предотвратит вложенность объектов без проверки посредством `isinstance`. Обсуждение других вариантов в данной области ищите в руководствах по Python; скажем, классы могут также возвращать для неподдерживаемых операндов специальный объект `NotImplemented`, чтобы влиять на выбор методов (это трактуется, как будто бы метод не был определен).

Сложение на месте

Чтобы реализовать дополненное сложение на месте (+=), понадобится написать код либо `__iadd__`, либо `__add__`. Второй метод используется, если отсутствует первый. На самом деле по указанной причине классы `Commuter` из предыдущего раздела уже поддерживают операцию +=: интерпретатор Python выполняет `__add__` и присваивает результат. Однако метод `__iadd__` позволяет более эффективно реализовывать изменения на месте, где это применимо:

```
>>> class Number:
        def __init__(self, val):
            self.val = val
        def __iadd__(self, other):    # Явный метод __iadd__: x += y
            self.val += other         # Обычно возвращает self
            return self

>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
```

Для изменяемых объектов метод `__iadd__` часто можно специализировать для выполнения более быстрых изменений на месте:

```
>>> y = Number([1])                   # Изменение на месте быстрее, чем +
>>> y += [2]
>>> y += [3]
>>> y.val
[1, 2, 3]
```

Нормальный метод `__add__` выполняется в качестве запасного варианта, но может быть не в состоянии оптимизировать случаи на месте:

```
>>> class Number:
        def __init__(self, val):
            self.val = val
        def __add__(self, other):    # Запасной вариант __add__: x = (x + y)
            return Number(self.val + other)     # Распространяет тип класса

>>> x = Number(5)
>>> x += 1
>>> x += 1                           # И += здесь выполняет конкатенацию
>>> x.val
7
```

Несмотря на то что внимание в примере было сосредоточено на операции +, имейте в виду, что каждая бинарная операция имеет похожие методы перегрузки для случаев с правой стороны и на месте, которые работают аналогично (например, `__mul__`, `__rmul__` и `__imul__`). Тем не менее, правосторонние методы являются сложной темой и менее часто используются на практике; вы реализуете их лишь тогда, когда операции должны быть коммутативными, и только если вообще нужна поддержка таких операций. Скажем, класс `Vector` (вектор) может применять эти инструменты, но класс `Employee` (сотрудник) или `Button` (кнопка) – вероятно нет.

Выражения вызовов: `__call__`

Перейдем к нашему следующему методу перегрузки: метод `__call__` вызывается при вызове вашего экземпляра. Нет, это вовсе не циклическое определение – если метод `__call__` определен, то Python выполняет его для выражений вызова функций, применяемых к вашему экземпляру, с передачей ему любых позиционных или ключевых аргументов, которые были отправлены. Это позволяет экземплярам соответствовать API-интерфейсу на основе функций:

```
>>> class Callee:
        def __call__(self, *pargs, **kargs): # Перехватывает вызовы экземпляра
            print('Called:', pargs, kargs)   # Принимает произвольные аргументы

>>> C = Callee()
>>> C(1, 2, 3)                               # C - вызываемый объект
Called: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Called: (1, 2, 3) {'y': 5, 'x': 4}
```

Говоря более формально, метод `__call__` поддерживает все режимы передачи аргументов, которые были исследованы в главе 18 первого тома – все, что было передано экземпляру, передается методу `__call__` вместе обычным подразумеваемым аргументом экземпляра. Скажем, определения метода:

```
class C:
    def __call__(self, a, b, c=5, d=6): ...  # Нормальные аргументы и аргументы
                                             #   со стандартными значениями
class C:
    def __call__(self, *pargs, **kargs): ...   # Сбор произвольных аргументов

class C:
    def __call__(self, *pargs, d=6, **kargs): ... # Аргумент с передачей только
                                                  # по ключевым словам Python 3.X
```

соответствуют следующим вызовам экземпляра:

```
X = C()
X(1, 2)                          # Стандартные значения не указаны
X(1, 2, 3, 4)                    # Позиционные аргументы
X(a=1, b=2, d=4)                 # Ключевые аргументы
X(*[1, 2], **dict(c=3, d=4))     # Распаковка произвольных аргументов
X(1, *(2,), c=3, **dict(d=4))    # Смешанные режимы
```

Чтобы освежить в памяти тему аргументов функций, обратитесь в главу 18 первого тома. Совокупный эффект в том, что классы и экземпляры с методом `__call__` поддерживают точно такие же синтаксис и семантику аргументов, как нормальные функции и методы.

Перехват выражений вызовов вроде показанных выше позволяет экземплярам класса эмулировать внешний вид и поведение сущностей вроде функций, а также предохраняет информацию о состоянии для использования во время вызовов. При исследовании областей видимости в главе 17 первого тома приводился пример, подобный представленному далее, но теперь вы должны достаточно знать о перегрузке операций, чтобы лучше понимать данную схему:

```
>>> class Prod:
        def __init__(self, value):  # Принимает всего один аргумент
            self.value = value
        def __call__(self, other):
            return self.value * other

>>> x = Prod(2)                     # "Запоминает" 2 в состоянии
>>> x(3)                            # 3 (переданное значение) * 2 (состояние)
6
>>> x(4)
8
```

Реализация `__call__` в этом примере на первый взгляд может показаться несколько неоправданной. Простой метод способен обеспечить похожий результат:

```
>>> class Prod:
        def __init__(self, value):
            self.value = value
        def comp(self, other):
            return self.value * other

>>> x = Prod(3)
>>> x.comp(3)
9
>>> x.comp(4)
12
```

Однако метод `__call__` может стать более полезным при взаимодействии с API-интерфейсами (т.е. библиотеками), ожидающими функций — он позволяет реализовывать объекты, которые соответствуют ожидаемому интерфейсу вызова функций, но также предохраняют информацию о состоянии и другие активы классов, такие как наследование. Фактически `__call__` можно считать третьим по частоте использования методом перегрузки операций после конструктора `__init__` и альтернативных форматов отображения `__str__` и `__repr__`.

Функциональные интерфейсы и код, основанный на обратных вызовах

В качестве примера отметим, что комплект инструментов для построения графических пользовательских инструментов `tkinter` (`Tkinter` в Python 2.X) позволяет регистрировать функции как обработчики событий (так называемые *обратные вызовы*) — когда возникают события, `tkinter` вызывает зарегистрированные объекты. Если вы хотите, чтобы обработчик событий сохранял состояние между событиями, тогда можете зарегистрировать либо *связанный метод* класса, либо экземпляр, который с помощью `__call__` обеспечивает соответствие ожидаемому интерфейсу.

Скажем, в коде предыдущего раздела `x.comp` из второго примера и `x` из первого могут таким способом передавать объекты, подобные функциям. *Функции замыканий* с состоянием из объемлющих областей видимости, рассматриваемые в главе 17 первого тома, способны достигать похожего эффекта, но не обеспечивают столько поддержки для множества операций или настройки.

В следующей главе связанные методы обсуждаются более подробно, а пока ниже представлен гипотетический пример метода `__call__` применительно к области графических пользовательских интерфейсов. Приведенный далее класс определяет объект, поддерживающий интерфейс вызова функций, но также запоминающий цвет, который должна получить кнопка при щелчке на ней в будущем:

```
class Callback:
    def __init__(self, color):              # Функция + информация о состоянии
        self.color = color
    def __call__(self):                     # Поддерживает вызовы без аргументов
        print('turn', self.color)
```

В контексте графического пользовательского интерфейса мы можем зарегистрировать экземпляры этого класса в качестве обработчиков событий для кнопок, хотя графический пользовательский интерфейс рассчитывает на возможность вызова обработчиков событий как простых функций без аргументов:

```
# Обработчики
cb1 = Callback('blue')                      # Запомнить blue
cb2 = Callback('green')                     # Запомнить green

B1 = Button(command=cb1)                    # Зарегистрировать обработчики
B2 = Button(command=cb2)
```

Когда позже на кнопке совершается щелчок, объект экземпляра вызывается как простая функция без аргументов подобно показанным ниже вызовам. Тем не менее, поскольку этот объект сохраняет состояние в виде атрибутов экземпляра, он запоминает, что делать, становясь объектом *функции с состоянием*:

```
# События
cb1()                                       # Выводит turn blue
cb2()                                       # Выводит turn green
```

В действительности многие считают такие классы лучшим способом предохранения информации о состоянии в языке Python (во всяком случае, согласно общепринятым принципам стиля Python). С помощью ООП запоминание состояния делается явным посредством присваивания значений атрибутам. Данный способ отличается от других методик предохранения состояния (например, глобальные переменные, ссылки из объемлющих областей видимости и изменяемые аргументы со стандартными значениями), которые полагаются на более ограниченное или менее явное поведе-

ние. Кроме того, добавленная структура и настройка в классах выходит за рамки одного лишь предохранения состояния.

С другой стороны, в базовых ролях предохранения состояния полезны также инструменты, подобные функциям замыканий, и оператор nonlocal из Python 3.X делает объемлющие области видимости жизнеспособной альтернативой во многих программах. Мы возвратимся к обсуждению таких компромиссов, когда приступим к реализации реальных декораторов в главе 39, но ниже показан эквивалент в форме *замыкания*:

```
def callback(color):           # Объемлющая область видимости или атрибуты
    def oncall():
        print('turn', color)
    return oncall

cb3 = callback('yellow')       # Обработчик, подлежащий регистрации
cb3()                          # При возникновении события выводит turn yellow
```

Прежде чем двигаться дальше, рассмотрим два других способа, которыми программисты на Python иногда привязывают информацию к функции обратного вызова подобного рода. Один вариант предусматривает использование аргументов со стандартными значениями в функциях lambda:

```
cb4 = (lambda color='red': 'turn ' + color)  # Стандартные значения тоже
                                             #   предохраняют состояние
print(cb4())
```

Второй вариант предполагает применение *связанных методов* класса, которые мы кратко представим здесь. Объект связанного метода – это разновидность объекта, запоминающего экземпляр self и функцию, на которую он ссылается. Такой объект впоследствии может быть вызван как простая функция без указания экземпляра:

```
class Callback:
    def __init__(self, color):       # Класс с информацией о состоянии
        self.color = color
    def changeColor(self):           # Нормальный именованный метод
        print('turn', self.color)

cb1 = Callback('blue')
cb2 = Callback('yellow')

B1 = Button(command=cb1.changeColor) # Связанный метод: ссылка, не вызывается
B2 = Button(command=cb2.changeColor) # Запоминает пару функция + экземпляр self
```

В данном случае, когда позже на кнопке производится щелчок для имитации его выполнения в графическом пользовательском интерфейсе, вместо самого экземпляра вызывается его метод changeColor, который обработает информацию о состоянии объекта:

```
cb1 = Callback('blue')
obj = cb1.changeColor       # Зарегистрированный обработчик событий
obj()                       # При возникновении события выводит turn blue
```

Обратите внимание, что функция lambda здесь не требуется, т.к. ссылка на связанный метод сама по себе уже откладывает вызов на потом. Такая методика проще, но вероятно менее универсальна, чем перегрузка вызовов с помощью __call__. Связанные методы более подробно обсуждаются в следующей главе.

Вы увидите еще один пример с методом __call__ в главе 32, где мы будем его использовать для реализации того, что известно как *декоратор функции* – вызываемый

объект, часто применяемый для добавления уровня логики поверх внедренной функции. Поскольку метод `__call__` позволяет присоединять к вызываемому объекту информацию о состоянии, он является естественной методикой реализации для функции, которая должна помнить о вызове еще одной функции, когда она вызывается сама. За дополнительными примерами работы с методом `__call__` обращайтесь к вариантам сохранения состояния в главе 17 первого тома, а также к более развитым декораторам и метаклассам в главах 39 и 40.

Сравнения: `__lt__`, `__gt__` и другие

Следующая группа методов перегрузки поддерживает сравнения. Как было указано в табл. 30.1, классы могут определять методы для перехвата всех шести операций сравнения: <, >, <=, >=, == и !=. В целом эти методы использовать легко, но необходимо иметь в виду описанные далее ограничения.

- В отличие от обсуждаемой ранее пары `__add__`/`__radd__` правосторонних вариантов методов сравнения не существует. Когда сравнение поддерживает только один операнд, взамен применяются зеркальные методы (скажем, `__lt__` и `__gt__` зеркальны относительно друг друга).
- Явных взаимоотношений между операциями сравнения не существует. Например, истинность операции == вовсе не подразумевает, что != даст ложь, поэтому для корректного поведения обеих операций должны быть определены оба метода `__eq__` и `__ne__`.
- В Python 2.X метод `__cmp__` используется для всех сравнений, если не определены более специфические методы сравнений; он возвращает число, которое меньше, равно или больше нуля, соответствующее результату сравнения его двух аргументов (`self` и второй операнд). Для вычисления своего результата метод `__cmp__` часто применяет встроенную функцию `cmp(x, y)`. В Python 3.X метод `__cmp__` и встроенная функция `cmp` были удалены: используйте вместо них более специфические методы.

Из-за нехватки места мы не можем провести глубокие исследования методов сравнений, но в качестве краткого введения взгляните на следующий класс и тестовый код:

```
class C:
    data = 'spam'
    def __gt__(self, other):          # Версия для Python 3.X и 2.X
        return self.data > other
    def __lt__(self, other):
        return self.data < other

X = C()
print(X > 'ham')                      # True (выполняется __gt__)
print(X < 'ham')                      # False (выполняется __lt__)
```

При запуске под управлением Python 3.X или 2.X операции `print` в конце отображают ожидаемые результаты, отмеченные в комментариях, поскольку методы класса перехватывают и реализуют выражения сравнений. Дополнительные детали ищите в руководствах по Python и в других справочных ресурсах; например, `__lt__` применяется для сортировок в Python3.X, а что касается бинарных операций выражений, то эти методы могут также возвращать объект `NotImplemented` для неподдерживаемых аргументов.

Метод __cmp__ в Python 2.X

В Python 2.X метод __cmp__ используется как запасной вариант, если не определены более специфические методы: его целочисленный результат применяется для оценки выполняемой операции. Например, следующий код дает в Python 2.X тот же самый результат, что и код из предыдущего раздела, но терпит неудачу в Python 3.X, т.к. метод __cmp__ больше не используется:

```
class C:
    data = 'spam'                    # Только в Python 2.X
    def __cmp__(self, other):        # В Python 3.X метод __cmp__ не используется
        return cmp(self.data, other) # Функция cmp в Python 3.X не определена

X = C()
print(X > 'ham')                     # True (выполняется __cmp__)
print(X < 'ham')                     # False (выполняется __cmp__)
```

Обратите внимание, что этот код терпит неудачу в Python 3.X из-за того, что метод __cmp__ перестал быть специальным, а не потому, что встроенной функции cmp больше не существует. Если мы изменим код предыдущего класса, как показано ниже, чтобы попытаться смоделировать вызов cmp, то код по-прежнему будет работать в Python 2.X, но давать отказ в Python 3.X:

```
class C:
    data = 'spam'
    def __cmp__(self, other):
        return (self.data > other) - (self.data < other)
```

У вас может возникнуть вопрос: для чего рассматривать метод сравнения, который больше не поддерживается в Python 3.X? Хотя легче полностью забыть историю, настоящая книга задумывалась для читателей, применяющих обе линейки, Python 2.X и Python 3.X. Поскольку метод __cmp__ может встречаться в коде Python 2.X, который читателям придется использовать или сопровождать, он заслуживает рассмотрения в книге. Кроме того, удаление метода __cmp__ было большей неожиданностью, чем изъятие описанного ранее метода __getslice__, так что он мог применяться дольше. Однако если вы используете Python 3.X или заинтересованы в будущем запуске своего кода под управлением Python 3.X, тогда больше не применяйте __cmp__: взамен отдавайте предпочтение более специфическим методам сравнений.

Булевские проверки: __bool__ и __len__

Следующий набор методов действительно полезен. Как уже известно, каждый объект в Python по своему существу является истинным или ложным. При написании кода классов вы можете определять, что это означает для объектов, реализуя методы, которые дают значения True или False экземпляров по запросу. Имена таких методов отличаются в линейках Python; здесь мы начнем с Python 3.X, после чего представим эквивалент в Python 2.X.

Ранее кратко упоминалось, что в булевских контекстах Python сначала пробует получить прямое булевское значение с помощью метода __bool__; если данный метод отсутствует, то Python посредством __len__ пытается вывести значение истинности из длины объекта. Первый метод для получения булевского значения обычно использует состояние объекта и другую информацию. Вот как он применяется в Python 3.X:

```
>>> class Truth:
        def __bool__(self): return True

>>> X = Truth()
>>> if X: print('yes!')

yes!

>>> class Truth:
        def __bool__(self): return False

>>> X = Truth()
>>> bool(X)
False
```

Если метод `__bool__` отсутствует, тогда Python прибегает к методу `__len__`, т.к. непустой объект считается истинным (т.е. ненулевая длина означает, что объект истинный, а нулевая – что он ложный):

```
>>> class Truth:
        def __len__(self): return 0

>>> X = Truth()
>>> if not X: print('no!')

no!
```

Если присутствуют *оба* метода, то Python отдает предпочтение `__bool__` перед `__len__`, потому что он более специфичен:

```
>>> class Truth:
        def __bool__(self): return True # Python 3.X пробует сначала __bool__
        def __len__(self): return 0     # Python 2.X пробует сначала __len__

>>> X = Truth()
>>> if X: print('yes!')

yes!
```

Если ни один из двух методов истинности не определен, тогда объект бессодержательно считается истинным (хотя любые потенциальные последствия для склонных к метафизике читателей полностью случайны):

```
>>> class Truth:
        pass

>>> X = Truth()
>>> bool(X)
True
```

Во всяком случае, так выглядит `Truth` в Python 3.X. В Python 2.X приведенные примеры не генерируют исключения, но некоторые из производимых ими результатов могут показаться несколько странными (и стать причиной жизненного кризиса или даже двух), если только вы не прочитаете следующий раздел.

Булевские методы в Python 2.X

Увы, ситуация не настолько драматична как было объявлено – во всем коде из предыдущего раздела пользователи Python 2.X просто используют `__nonzero__` вместо `__bool__`. Метод `__nonzero__` из Python 2.X в Python 3.X переименован на `__bool__`, но в остальном булевские проверки работают точно так же; в качестве запасного варианта в обеих линейках Python 3.X и Python 2.X применяется метод `__len__`.

Есть одна тонкость: если вы не используете имя из Python 2.X, то первая проверка в предыдущем разделе будет работать точно так же, но лишь потому, что `__bool__` не распознается как особое имя метода в Python 2.X, а объекты по умолчанию считаются истинными! Чтобы увидеть такое отличие между версиями вживую, необходимо возвратить `False`:

```
C:\code> c:\python37\python
>>> class C:
        def __bool__(self):
            print('in bool')
            return False

>>> X = C()
>>> bool(X)
in bool
False
>>> if X: print(99)

in bool
```

В Python 3.X код работает, как было заявлено. Тем не менее, в Python 2.X метод `__bool__` игнорируется и объект всегда по умолчанию расценивается как истинный:

```
C:\code> c:\python27\python
>>> class C:
        def __bool__(self):
            print('in bool')
            return False

>>> X = C()
>>> bool(X)
True
>>> if X: print(99)

99
```

Краткая история такова: в Python 2.X применяйте `__nonzero__` для булевских значений или возвращайте `0` из запасного метода `__len__`, чтобы указывать на ложное значение:

```
C:\code> c:\python27\python
>>> class C:
        def __nonzero__(self):
            print('in nonzero')
            return False # Возвращает целое число (или True/False, то же что 1/0)

>>> X = C()
>>> bool(X)
in nonzero
False
>>> if X: print(99)

in nonzero
```

Но имейте в виду, что метод `__nonzero__` работает только в Python 2.X. Если использовать `__nonzero__` в Python 3.X, то он молчаливо игнорируется, а объект по умолчанию классифицируется как истинный — в точности то, что происходит в случае применения в Python 2.X метода `__bool__` из Python 3.X!

И теперь, когда нас удалось перейти в область философии, давайте займемся последним контекстом перегрузки: *кончиной объектов*.

Уничтожение объектов: `__del__`

Пришло время завершать главу – и научиться делать то же самое с нашими объектами классов. Мы видели, как *конструктор* `__init__` вызывается всякий раз, когда генерируется экземпляр (и отметили, что сначала для создания объекта выполняется `__new__`). Его противоположность, метод *деструктора* `__del__`, запускается автоматически при возвращении пространства памяти, занимаемого экземпляром (т.е. во время "сборки мусора"):

```
>>> class Life:
        def __init__(self, name='unknown'):
            print('Hello ' + name)
            self.name = name
        def live(self):
            print(self.name)
        def __del__(self):
            print('Goodbye ' + self.name)

>>> brian = Life('Brian')
Hello Brian
>>> brian.live()
Brian
>>> brian = 'loretta'
Goodbye Brian
```

Когда объекту `brian` присваивается строка, ссылка на экземпляр `Life` утрачивается и потому запускается его метод деструктора. Это работает и может быть полезным для реализации действий по очистке, таких как закрытие подключения к серверу. Однако деструкторы не настолько часто используется в Python и в ряде других языков ООП по причинам, которые описаны в следующем разделе.

Замечания относительно использования деструкторов

Метод деструктора работает так, как документировано, но с ним связано несколько известных предостережений и откровенно темных уголков, из-за которых он редко встречается в коде на Python.

- ***Необходимость.*** С одной стороны, деструкторы не настолько полезны в Python, как в ряде других языков ООП. Поскольку Python автоматически возвращает все пространство памяти, занимаемое экземпляром, когда экземпляр уничтожается, деструкторы не требуются для управления памятью. В текущей реализации CPython также не нужно закрывать в деструкторах файловые объекты, удерживаемые экземпляром, потому что при уничтожении экземпляра они автоматически закрываются. Тем не менее, в главе 9 первого тома упоминалось, что иногда по-прежнему лучше в любом случае запускать методы закрытия файлов, т.к. поведение автоматического закрытия может варьироваться в альтернативных реализациях Python (скажем, в Jython).

- ***Предсказуемость.*** С другой стороны, не всегда легко спрогнозировать, когда экземпляр будет уничтожен. В ряде случаев внутри системных таблиц могут присутствовать долго существующие ссылки на ваши объекты, которые препятствуют выполнению деструкторов, когда программа ожидает их запуска. Вдобавок Python вовсе не гарантирует, что методы деструкторов будут вызваны для объектов, которые все еще существуют при завершении работы интерпретатора.

- *Исключения.* На самом деле метод `__del__` может оказаться сложным в применении даже по более тонким причинам. Например, возникающие в нем исключения просто выводят предупреждающее сообщение в `sys.stderr` (стандартный поток ошибок), а не генерируют событие исключения, из-за непредсказуемого контекста, в котором метод `__del__` выполняется сборщиком мусора – не всегда возможно знать, куда такое исключение должно быть доставлено.

- *Циклы.* Кроме того, циклические (круговые) ссылки между объектами могут препятствовать запуску сборки мусора, когда она ожидается. По умолчанию включенный необязательный обнаружитель циклов способен со временем автоматически собирать объекты подобного рода, но только если они не имеют методов `__del__`. Поскольку это относительно малоизвестно, здесь мы проигнорируем дальнейшие детали; за дополнительной информацией обращайтесь к описанию метода `__del__` и модуля сборщика мусора `gc` в стандартных руководствах по Python.

Из-за таких недостатков часто лучше реализовывать действия завершения в явно вызываемом методе (скажем, `shutdown`). Как будет описано в следующей части книги, оператор `try/ finally` также поддерживает действия завершения подобно оператору `with` для объектов, которые поддерживают модель диспетчеров контекстов.

Резюме

Здесь было приведено столько примеров перегрузки, сколько позволил объем главы. Большинство оставшихся методов перегрузки операций работают подобно исследованным в этой главе, и все они являются просто привязками для перехвата операций над встроенными типами. Например, некоторые методы имеют уникальные списки аргументов или возвращаемые значения, но общая схема использования остается такой же. Позже в книге мы увидим несколько других методов перегрузки в действии:

- в главе 34 применяются методы `__enter__` и `__exit__` в диспетчерах контекстов операторов `with`;
- в главе 38 используются методы извлечения/установки `__get__` и `__set__` дескрипторов классов;
- в главе 40 применяется метод создания объектов `__new__` в контексте метаклассов.

Вдобавок некоторые изученные здесь методы, такие как `__call__` и `__str__`, будут задействованы в примерах позже в книге. Однако за более полным описанием обращайтесь к другим источникам документации – дополнительные детали о других методах перегрузки ищите в стандартном руководстве или в различных справочниках по языку Python.

В следующей главе мы оставим область механики классов, чтобы исследовать общепринятые паттерны проектирования – распространенные способы использования и объединения классов, направленные на оптимизацию многократного применения кода. Затем мы рассмотрим несколько продвинутых тем и перейдем к исключениям – последней основной теме книги. Тем не менее, прежде чем двигаться дальше, ответьте на контрольные вопросы главы, закрепив знания приведенных в главе концепций.

Проверьте свои знания: контрольные вопросы

1. Какие два метода перегрузки операций можно использовать для поддержки итерации в классах?
2. Какие два метода перегрузки операций обрабатывают вывод, и в каких контекстах?
3. Как можно перехватывать операции нарезания в классе?
4. Как можно перехватывать сложение на месте в классе?
5. Когда должна предоставляться перегрузка операций?

Проверьте свои знания: ответы

1. Классы могут поддерживать итерацию путем определения (или наследования) метода `__getitem__` или `__iter__`. Во всех итерационных контекстах Python сначала пытается применить метод `__iter__`, возвращающий объект, который поддерживает протокол итерации с помощью метода `__next__`: если поиск в иерархии наследования не привел к нахождению метода `__iter__`, тогда Python прибегает к методу индексирования `__getitem__`, многократно вызывая его с последовательно увеличивающимися индексами. В случае использования оператора `yield` метод `__next__` может быть создан автоматически.
2. Методы `__str__` и `__repr__` реализуют отображения объектов при выводе. Первый вызывается встроенными функциями `print` и `str`; второй вызывается `print` и `str`, если отсутствует `__str__`, и всегда вызывается встроенной функцией `repr`, при эхо-выводе в интерактивной подсказке и для вложенных появлений. То есть метод `__repr__` применяется везде, исключая `print` и `str`, когда определен метод `__str__`. Метод `__str__` обычно используется для отображений, дружественных к пользователю, а `__repr__` предоставляет для объекта дополнительные детали или форму как в коде.
3. Нарезание перехватывается методом индексирования `__getitem__`: он вызывается с объектом среза, а не с простым целочисленным индексом, и при необходимости объекты срезов можно передавать или ожидать. В Python 2.X может также применяться метод `__getslice__` (исчезнувший в Python 3.X) для срезов с двумя пределами.
4. Сложение на месте сначала пытается использовать метод `__iadd__` и затем `__add__` с присваиванием. Та же схема применяется для всех бинарных операций. Для правостороннего сложения также доступен метод `__radd__`.
5. Когда класс естественным образом согласуется с интерфейсами встроенного типа или должен их эмулировать. Например, коллекции могут имитировать интерфейсы последовательностей или отображений, а вызываемые объекты могут быть реализованы для использования с API-интерфейсом, который ожидает функцию. Однако в целом вы не должны реализовывать операции выражений, если они естественно и логически не подходят для ваших объектов — взамен применяйте нормально именованные методы.

ГЛАВА 31

Проектирование с использованием классов

До сих пор в текущей части книги внимание было сосредоточено на применении инструмента ООП на языке Python — класса. Но ООП также связано с *вопросами проектирования*, т.е. с тем, каким образом использовать классы для моделирования полезных объектов. В этой главе мы затронем несколько основных идей ООП и представим ряд дополнительных примеров, более реалистичных, чем многие показанные ранее.

Попутно мы реализуем некоторые распространенные паттерны проектирования в ООП на Python, такие как наследование, композиция, делегирование и фабрики. Мы также исследуем концепции классов, ориентированные на проектирование, вроде псевдозакрытых атрибутов, множественного наследования и связанных методов.

Одно заблаговременное примечание: некоторые упоминаемые здесь термины, относящиеся к проектированию, требуют большего объема пояснений, чем я смог предложить в книге. Если данный материал вызвал у вас интерес, тогда я рекомендую в качестве следующего шага почитать какую-нибудь книгу по объектно-ориентированному проектированию или паттернам проектирования. Как вы вскоре убедитесь, хорошая новость заключается в том, что Python делает многие традиционные паттерны проектирования тривиальными.

Python и объектно-ориентированное программирование

Давайте начнем с обзора — реализация ООП в языке Python может быть сведена к трем идеям.

Наследование

Наследование основано на поиске атрибутов в Python (в выражениях `X.name`).

Полиморфизм

В выражении `X.method` смысл `method` зависит от типа (класса) подчиненного объекта `X`.

Инкапсуляция

Методы и операции реализуют поведение, хотя сокрытие данных по умолчанию является соглашением.

К этому времени вы должны иметь хорошее представление о том, что такое наследование в Python. Ранее мы также несколько раз обсуждали полиморфизм в Python; он вытекает из отсутствия объявлений типов. Поскольку атрибуты всегда распознаются во время выполнения, объекты, которые реализуют те же самые интерфейсы, автоматически становятся взаимозаменяемыми; клиенты не обязаны знать о том, какие виды объектов реализуют методы, вызываемые клиентами.

Инкапсуляция означает пакетирование в Python, т.е. сокрытие деталей реализации за интерфейсом объекта. Она вовсе не означает принудительную защиту, несмотря на возможность ее реализации с помощью кода, как мы увидим в главе 39. Однако инкапсуляция доступна и полезна в Python: она позволяет изменять реализацию интерфейса объекта, не влияя на пользователей этого объекта.

Полиморфизм означает интерфейсы, а не сигнатуры вызовов

Некоторые языки ООП определяют полиморфизм как подразумевающий перегрузку функций на основе сигнатур типов их аргументов — количестве и/или типах переданных аргументов. Ввиду отсутствия объявлений типов в Python такая концепция фактически неприменима; как будет показано, полиморфизм в Python базируется на *интерфейсах* объектов, а не на типах.

Если вы тоскуете по тем дням, когда программировали на C++, то можете попробовать перегрузить методы посредством списков их аргументов, примерно так:

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

Код запустится, но из-за того, что def просто присваивает объект имени в области видимости класса, останется лишь одно определение функции метода — *последнее*. Иными словами, ситуация такая же, как если бы вы ввели X = 1 и затем X = 2; имя X получило бы значение 2. Следовательно, может существовать только одно определение имени метода.

В случае реальной необходимости вы всегда можете написать код выбора на основе типов, используя идеи проверки типов, с которыми мы встречались в главах 4 и 9 первого тома, либо инструменты списков аргументов из главы 18 первого тома:

```
class C:
    def meth(self, *args):
        if len(args) == 1:          # Ветвление по количеству аргументов
            ...
        elif type(arg[0]) == int:   # Ветвление по типам аргументов
                                    #   (или isinstance())
            ...
```

Тем не менее, обычно вы не должны поступать так — это не стиль Python. Как было описано в главе 16 первого тома, вы обязаны писать код в расчете только на *интерфейс* объекта, но не на специфический *тип* данных. В итоге код будет полезен для более широкой категории типов и приложений, как сейчас, так и в будущем:

```
class C:
    def meth(self, x):
        x.operation()    # Предположим, что x делает что-то правильное
```

Также в целом считается лучше применять отличающиеся *имена* методов для отдельных операций, а не полагаться на сигнатуры вызовов (не имеет значения, на каком языке вы пишете код).

Хотя объектная модель Python прямолинейна, основное мастерство в ООП проявляется в способе объединения классов для достижения целей программы. В следующем разделе начинается обзор особенностей использования классов более крупными программами для извлечения из них выгоды.

Объектно-ориентированное программирование и наследование: отношения "является"

Мы уже довольно глубоко исследовали механику наследования, но сейчас я хотел бы показать вам пример того, как оно может применяться для моделирования взаимосвязей из реального мира. С точки зрения *программиста* наследование вводится уточнениями атрибутов, которые инициируют поиск имен в экземплярах, в их классах и затем в любых суперклассах. С точки зрения *проектировщика* наследование является способом указания членства в наборе: класс определяет набор свойств, которые могут наследоваться и настраиваться более специфическими наборами (т.е. подклассами).

В целях иллюстрации давайте создадим робота по приготовлению пиццы, о котором шла речь в начале данной части книги. Предположим, мы решили проверить альтернативные пути карьерного продвижения и открыть пиццерию (совсем неплохо складывается карьера). Одним из первых дел, которые понадобится выполнить, будет наем персонала для обслуживания клиентов, готовки еды и т.д. Будучи в душе инженерами, мы решили построить робота для приготовления пиццы, но по причине своей политической и кибернетической корректности мы также решили сделать нашего робота полноправным сотрудником с заработной платой.

Наша команда из пиццерии может быть определена с помощью четырех классов, находящихся в файле примера `employees.py` для Python 3.X и 2.X, содержимое которого приведено ниже. Наиболее универсальный класс, `Employee`, предоставляет общее поведение, такое как повышение зарплат (`giveRaise`) и вывод (`__repr__`). Есть два вида сотрудников и потому два подкласса `Employee` — `Chef` (шеф-повар) и `Server` (официант). В обоих подклассах переопределяется метод `work`, чтобы выводить более специфические сообщения. Наконец, наш робот по приготовлению пиццы моделируется даже более специфическим классом — `PizzaRobot` представляет собой разновидность класса `Chef`, который является видом `Employee`. В терминах ООП мы называем такие отношения связями "является": робот является шеф-поваром, который является сотрудником. Вот что находится в файле `employees.py`:

```
# Файл employees.py (Python 2.X + 3.X)
from __future__ import print_function

class Employee:
    def __init__(self, name, salary=0):
        self.name = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print(self.name, "does stuff")                # делает что-то
    def __repr__(self):
        return "<Employee: name=%s, salary=%s>" % (self.name, self.salary)
```

```
class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, "makes food")              # готовит еду

class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print(self.name, "interfaces with customer") # взаимодействует
                                                     # с клиентом

class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print(self.name, "makes pizza")              # готовит пиццу

if __name__ == "__main__":
    bob = PizzaRobot('bob')      # Создать робота по имени bob
    print(bob)                   # Выполняется унаследованный метод __repr__
    bob.work()                   # Выполняется действие, специфичное для типа
    bob.giveRaise(0.20)          # Повысить зарплату роботу bob на 20%
    print(bob); print()

    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work()
```

Когда мы запускаем код самотестирования, включенный в этот модуль, создается робот для приготовления пиццы по имени bob, который наследует имена от трех классов: PizzaRobot, Chef и Employee. Например, при выводе bob выполняется метод Employee.__repr__, а при повышении заработной платы bob вызывается метод Employee.giveRaise, потому что его находит процедура поиска в иерархии наследования:

```
c:\code> python employees.py
<Employee: name=bob, salary=50000>
bob makes pizza
<Employee: name=bob, salary=60000.0>

Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

В иерархии классов подобного рода вы всегда можете создавать экземпляры любых классов, а не только тех, которые расположены в нижней части дерева. Скажем, внутри цикла for в коде самотестирования данного модуля создаются экземпляры всех четырех классов; каждый реагирует по-разному при вызове метода work, т.к. реализации этого метода во всех классах отличаются. Например, робот bob получает метод work от самого специфичного (т.е. находящегося ниже всех) класса PizzaRobot.

Разумеется, классы лишь *эмулируют* объекты реального мира. В настоящий момент метод work выводит сообщение, но позже его можно было бы расширить для выполнения реальной работы (если вы воспринимаете данный раздел слишком буквально, тогда взгляните на интерфейсы Python к устройствам, таким как последовательные порты, платы микроконтроллеров Arduino и одноплатные микрокомпьютеры Raspberry Pi).

Объектно-ориентированное программирование и композиция: отношения "имеет"

Понятие композиции было введено в главах 26 и 28. С точки зрения *программиста* композиция затрагивает внедрение других объектов в объект контейнера и их активизацию для реализации методов контейнера. С точки зрения *проектировщика* композиция является еще одним способом представления отношений в предметной области. Но вместо членства в наборе композиция имеет дело с компонентами — частями целого.

Композиция также отражает взаимосвязи между частями, называемые отношениями "имеет". В некоторых книгах по объектно-ориентированному проектированию на композицию ссылаются как на *агрегирование* или проводят различие между этими двумя терминами, используя агрегирование для описания более слабой зависимости между контейнером и его содержимым. Здесь под "композицией" понимается просто совокупность внедренных объектов. Составной класс обычно предоставляет собственный интерфейс и реализует его, направляя выполнение действий внедренным объектам.

После реализации классов сотрудников давайте поместим их в пиццерию и предоставим работу. Наша пиццерия является составным объектом: в ней есть духовой шкаф, а также сотрудники вроде официантов и шеф-поваров. Когда клиент входит и размещает заказ, компоненты пиццерии приступают к действиям — официант принимает заказ, шеф-повар готовит пиццу и т.д. Следующий пример (файл `pizzashop.py`) выполняется одинаково в Python 3.X и 2.X и эмулирует все объекты и отношения в описанном сценарии:

```
# Файл pizzashop.py (Python 2.X + 3.X)
from __future__ import print_function
from employees import PizzaRobot, Server

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print(self.name, "orders from", server)        # заказы от
    def pay(self, server):
        print(self.name, "pays for item to", server)   # плата за единицу
class Oven:
    def bake(self):
        print("oven bakes")                 # духовой шкаф выпекает
class PizzaShop:
    def __init__(self):
        self.server = Server('Pat')         # Внедрить другие объекты
        self.chef = PizzaRobot('Bob')       # Робот по имени bob
        self.oven = Oven()
    def order(self, name):
        customer = Customer(name)           # Активизировать другие объекты
        customer.order(self.server)         # Заказы клиента, принятые официантом
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)
if __name__ == "__main__":
    scene = PizzaShop()                     # Создать составной объект
    scene.order('Homer')                    # Эмулировать заказ клиента Homer
    print('...')
    scene.order('Shaggy')                   # Эмулировать заказ клиента Shaggy
```

Класс PizzaShop является контейнером и контроллером; его конструктор создает и внедряет экземпляры классов сотрудников, код которых написан в предыдущем разделе, а также экземпляр определенного в текущем разделе класса Oven. Когда в коде самотестирования модуля вызывается метод order класса PizzaShop, внедренным объектам предлагается выполнить их действия по очереди. Обратите внимание, что мы создаем новый объект Customer для каждого заказа и передаем внедренный объект Server методам Customer; клиенты приходят и уходят, но официант представляет собой часть составного объекта пиццерии. Также обратите внимание, что сотрудники по-прежнему участвуют в отношении наследования; композиция и наследование — работающие совместно инструменты.

После запуска модуля пиццерия обслужит два заказа — от клиента Homer и от клиента Shaggy:

```
c:\code> python pizzashop.py
Homer orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Pat, salary=40000>
...
Shaggy orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Pat, salary=40000>
```

Опять-таки это по большей части игрушечная эмуляция, но объекты и взаимодействия отражают работу составных объектов. В качестве эмпирического правила запомните: классы могут представлять почти любые объекты и отношения, которые удастся выразить с помощью предложения; просто замените *имена существительные* классами (скажем, Oven), а *глаголы* методами (например, bake), и вы получите первое приближение к проектному решению.

Снова об обработчиках потоков данных

В качестве примера композиции, который может оказаться чуть более осязаемым, чем роботы по приготовлению пиццы, вспомним обобщенную функцию обработчика потоков данных, частично реализованную во введении в ООП в главе 26:

```
def processor(reader, converter, writer):
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Вместо применения простой функции мы можем написать код класса, который для выполнения своей работы использует композицию, чтобы обеспечить бо́льшую структурированность и поддержку наследования. В файле streams.py со следующим содержимым для Python 3.X/2.X демонстрируется возможный способ реализации класса (в нем также видоизменяется одно имя метода, потому что мы действительно запустим этот код):

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
```

```
    def process(self):
        while True:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)

    def converter(self, data):
      assert False, 'converter must be defined'  # Или сгенерировать исключение
```

Класс Processor определяет метод converter и ожидает его заполнения подклассами; он является примером модели *абстрактных суперклассов*, обрисованной в главе 29 (оператор assert более подробно рассматривается в части VII и просто генерирует исключение, если результатом проверки оказывается False). Реализованные подобным образом объекты reader и writer внедряются внутрь экземпляра класса (*композиция*), и мы поставляем логику преобразования в подклассе, а не передаем ее функции converter (*наследование*). Вот как выглядит содержимое файла converters.py:

```
from streams import Processor

class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':
    import sys
    obj = Uppercase(open('trispam.txt'), sys.stdout)
    obj.process()
```

Здесь класс Uppercase наследует логику цикла обработки потоков данных (и все остальное, что может быть реализовано в суперклассах). Ему необходимо определить только то, что в нем уникально – логику преобразования данных. При запуске файла создается и выполняется экземпляр, который читает из файла trispam.txt и записывает в поток данных stdout эквивалент в верхнем регистре содержимого:

```
c:\code> type trispam.txt
spam
Spam
SPAM!

c:\code> python converters.py
SPAM
SPAM
SPAM!
```

Чтобы обрабатывать различные виды потоков, при создании экземпляра передавайте разные типы объектов. Ниже вместо потока данных применяется выходной файл:

```
C:\code> python
>>> import converters
>>> prog = converters.Uppercase(open('trispam.txt'), open('trispamup.txt', 'w'))
>>> prog.process()

C:\code> type trispamup.txt
SPAM
SPAM
SPAM!
```

Но, как предполагалось ранее, мы также могли бы передавать произвольные объекты, реализованные в виде классов, которые определяют обязательные интерфейсы с методами ввода и вывода. Далее приведен простой пример передачи класса средства записи, который помещает текст внутрь HTML-дескрипторов:

```
C:\code> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
        def write(self, line):
            print('<PRE>%s</PRE>' % line.rstrip())
>>> Uppercase(open('trispam.txt'), HTMLize()).process()
<PRE>SPAM</PRE>
<PRE>SPAM</PRE>
<PRE>SPAM!</PRE>
```

Если вы проследите поток управления рассмотренного примера, то заметите, что мы получаем преобразование в верхний регистр (путем наследования) *и* форматирование HTML (за счет композиции), хотя основной логике обработки в исходном суперклассе `Processor` ничего не известно ни об одном, ни о другом шаге. Код обработки интересуется только тем, чтобы средства записи имели метод `write` и определяли метод по имени `convert`; его не заботит, что делают упомянутые методы, когда вызываются. Такой полиморфизм и инкапсуляция логики лежат в основе мощи классов в Python.

В том виде, как есть, суперкласс `Processor` предлагает только цикл просмотра файлов. В более реалистичном проекте мы могли бы расширить его с целью поддержки дополнительных программных инструментов для подклассов и в процессе работы превратить `Processor` в полнофункциональный прикладной фреймворк. Написание такого инструмента один раз в суперклассе позволяет его многократно использовать во всех разрабатываемых программах. Даже в этом простом примере из-за того, что благодаря классам было настолько много пакетировано и унаследовано, нам пришлось написать код лишь для шага форматирования HTML, а остальное досталось бесплатно.

За еще одним примером композиции в работе обратитесь к упражнению 9 в конце главы 32 и его решению в приложении Г; он похож на пример с пиццерией. В этой книге мы сосредоточили внимание на наследовании, потому что оно является главным инструментом, который сам язык предлагает для ООП. Но на практике композиция может применяться наравне с наследованием в качестве способа структурирования классов, особенно в более крупных системах. Мы видели, что наследование и композиция часто являются дополняющими друг друга (а временами и взаимоисключающими) методиками. Однако поскольку композиция относится к вопросам проектирования, выходящим за рамки языка Python и самой книги, за дополнительными сведениями по данной теме я отсылаю вас к другим ресурсам.

Что потребует внимания: классы и постоянство

В текущей части книги я несколько раз упоминал о поддержке постоянства объектов с помощью модулей `pickle` и `shelve` в Python, т.к. она особенно хорошо работает с экземплярами классов. На самом деле указанные инструменты оказываются достаточно убедительными для того, чтобы вообще использовать классы — организуя обработку посредством `pickle` и `shelve` экземпляра класса, мы получаем хранилище, которое содержит объединенные вместе данные и логику.

Скажем, помимо того, что разработанные в главе классы пиццерии позволяют эмулировать взаимодействия реального мира, они могли бы также служить основой

базы данных с постоянным хранением. С применением модулей pickle или shelve из Python экземпляры классов можно сохранять на диске за один шаг. Мы использовали shelve для хранения экземпляров классов в учебном руководстве по ООП в главе 28, но интерфейс pickle также удивительно легко применять с объектами:

```
import pickle
object = SomeClass()
file = open(filename, 'wb')     # Создать внешний файл
pickle.dump(object, file)       # Сохранить объект в файле

import pickle
file = open(filename, 'rb')
object = pickle.load(file)      # Извлечь объект в более позднее время
```

Модуль pickle преобразует объекты из памяти в сериализированные потоки байтов (строки в Python), которые можно хранить в файлах, отправлять по сети и т.д.; распаковка с помощью pickle преобразует потоки байтов в идентичные объекты в памяти. Модуль shelve похож, но сохраняет объекты, автоматически обработанные pickle, в базе данных с доступом по ключу, которая экспортирует словарный интерфейс:

```
import shelve
object = SomeClass()
dbase = shelve.open(filename)
dbase['key'] = object          # Сохранить под ключом

import shelve
dbase = shelve.open(filename)
object = dbase['key']          # Извлечь объект в более позднее время
```

В нашем примере с пиццерией использование классов для моделирования сотрудников означает возможность создания простой базы данных за счет выполнения небольшой дополнительной работы. Сохранение таких объектов экземпляров в файле с помощью pickle делает их постоянными между запусками программы на Python:

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.server, shop.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> import pickle
>>> pickle.dump(shop, open('shopfile.pkl', 'wb'))
```

Здесь в файле сохраняется весь составной объект shop целиком. Чтобы позже поместить его в другой сеанс или программу, также достаточно одного шага. В действительности объекты, восстановленные подобным образом, предохраняют свое состояние и поведение:

```
>>> import pickle
>>> obj = pickle.load(open('shopfile.pkl', 'rb'))
>>> obj.server, obj.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)

>>> obj.order('LSP')
LSP orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
LSP pays for item to <Employee: name=Pat, salary=40000>
```

Это просто запускает эмуляцию в существующем виде, но мы могли бы расширить модель пиццерии, чтобы отслеживать запасы, выручку и т.п. — ее сохранение в файле после внесения изменений предохранит обновленное состояние. Дополнительные сведения о модулях pickle и shelve ищите в руководстве по стандартной библиотеке, а также в главах 9 (первого тома), 28 и 37.

Объектно-ориентированное программирование и делегирование: промежуточные объекты-оболочки

Наряду с наследованием и композицией программисты, занимающиеся ООП, часто говорят о *делегировании*, что обычно подразумевает объекты контроллеров с внедренными другими объектами, которым они передают запросы операций. Контроллеры могут заниматься административными действиями, такими как ведение журналов либо проверка достоверности доступа, добавляя дополнительные шаги к компонентам интерфейса или отслеживая активные экземпляры.

В известном смысле делегирование является особой формой композиции с единственным внедренным объектом, управляемым классом *оболочки* (иногда называемого *промежуточным* классом), который предохраняет большую часть или весь интерфейс внедренного объекта. Понятие промежуточных классов временами применяется и к другим механизмам, таким как вызовы функций; при делегировании нас интересуют промежуточные классы для *всех* линий поведения объекта, включая вызовы методов и прочие операции.

Такая концепция была введена через пример в главе 28, и в Python она часто реализуется с помощью метода `__getattr__`, рассмотренного в главе 30. Поскольку этот метод перегрузки операции перехватывает доступ к несуществующим атрибутам, класс оболочки может использовать `__getattr__` для маршрутизации произвольного доступа к внутреннему объекту. Из-за того, что метод `__getattr__` дает возможность маршрутизировать запросы атрибутов обобщенным образом, класс оболочки предохраняет интерфейс внутреннего объекта и сам может добавлять дополнительные операции.

В качестве обзора взгляните на содержимое файла `trace.py` (одинаково выполняющегося в Python 2.X и 3.X):

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Сохранить объект
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)              # Трассировать извлечение
        return getattr(self.wrapped, attrname)   # Делегировать извлечение
```

Вспомните из главы 30, что метод `__getattr__` получает имя атрибута в виде строки. В коде атрибут извлекается из внутреннего объекта по строковому имени посредством встроенной функции `getattr` — вызов `getattr(X,N)` похож на `X.N`, но `N` является выражением, вычисляемым в строку во время выполнения, а не переменной. На самом деле вызов `getattr(X,N)` подобен выражению `X.__dict__[N]`, но первый вариант также производит поиск в иерархии наследования, как и `X.N`, а второй — нет (за информацией об атрибуте `__dict__` обращайтесь в главу 22 первого тома и в главу 29).

Вы можете применять подход с классом оболочки, продемонстрированный в модуле, для управления доступом к любому объекту с атрибутами — спискам, словарям и даже классам и экземплярам. Здесь класс `Wrapper` просто выводит трассировочное сообщение при каждом доступе к атрибуту и делегирует запрос атрибута внутреннему объекту `wrapped`:

```
>>> from trace import Wrapper
>>> x = Wrapper([1, 2, 3])                # Создать оболочку для списка
>>> x.append(4)                           # Делегировать списковому методу
Trace: append
```

```
>>> x.wrapped                        # Вывести внутренний объект
[1, 2, 3, 4]

>>> x = Wrapper({'a': 1, 'b': 2})    # Создать оболочку для словаря
>>> list(x.keys())                   # Делегировать словарному методу
Trace: keys
['a', 'b']
```

Совокупный эффект заключается в дополнении целого интерфейса объекта wrapped добавочным кодом в классе Wrapper. Мы можем использовать его для регистрации вызовов методов в журналах, направления вызовов методов дополнительной или специальной логике, адаптации класса к новому интерфейсу и т.д.

В следующей главе мы снова вернемся к понятиям внутренних объектов и делегирования операций как одному из способов расширения встроенных типов. Если вы заинтересовались паттерном проектирования с делегированием, тогда также обратитесь к обсуждению в главах 32 и 39 *декораторов функций* — тесно связанной концепции, предназначенной для дополнения вызова специфической функции или метода, а не целого интерфейса объекта, и *декораторов классов*, которые служат способом автоматического добавления таких основанных на делегировании оболочек ко всем экземплярам класса.

Примечание, касающееся нестыковки версий. Как было показано в примере из главы 28, делегирование интерфейсов объектов общими *классами посредников* значительно изменяется в Python 3.X, когда внутренние объекты реализуют методы перегрузки операций. Формально это является отличительной особенностью *классов нового стиля* и может присутствовать также в коде Python 2.X, если такая возможность включена; согласно следующей главе в Python 3.X она обязательна и потому часто считается изменением Python 3.X.

В стандартных классах Python 2.X методы перегрузки операций, запускаемые встроенными операциями, маршрутизируются через обобщенные методы перехвата атрибутов вроде `__getattr__`. Например, вывод внутреннего объекта напрямую приводит к вызову упомянутого метода для метода `__repr__` или `__str__`, который затем передает вызов внутреннему объекту. Такая схема действует для `__iter__`, `__add__` и других методов операций, рассмотренных в предыдущей главе.

В Python 3.X подобное больше не происходит: вывод не запускает метод `__getattr__` (или родственный ему `__getattribute__`, который мы исследуем в следующей главе) и взамен применяется стандартное отображение. В Python 3.X классы нового стиля ищут методы, неявно вызываемые встроенными операциями, в классах и полностью пропускают нормальный поиск в экземплярах. Явные операции извлечения атрибутов имен направляются методу `__getattr__` одинаково в Python 2.X и 3.X, но поиск методов встроенных операций отличается аспектами, которые могут повлиять на ряд инструментов, основанных на делегировании.

В следующей главе мы рассмотрим указанную проблему как изменение классов нового стиля, а также увидим ее вживую в главах 38 и 39 в контексте управляемых атрибутов и декораторов. Пока просто примите к сведению, что для кодовых схем делегирования может потребоваться переопределение методов перегрузки операций в классах оболочек (вручную, с помощью инструментов или посредством суперклассов), если они используются внутренними объектами, и вы хотите, чтобы их перехватывали классы нового стиля.

Псевдозакрытые атрибуты классов

Помимо более масштабных целей проектные решения классов должны также принимать меры относительно использования имен. Скажем, в учебном примере из главы 28 мы отмечали, что методы, определенные внутри класса универсального инструмента, могут быть модифицированы подклассами, если к ним открыт доступ, и упомянули о компромиссах такой политики — наряду с поддержкой настройки и прямых вызовов методов они также открыты для случайных замещений.

В части V мы выяснили, что каждое имя, которому присваивается значение на верхнем уровне файла модуля, экспортируется. По умолчанию то же самое справедливо для классов — сокрытие данных является соглашением, а клиенты могут извлекать либо изменять атрибуты в любом классе или экземпляре, на который они имеют ссылку. На самом деле в терминах C++ все атрибуты "открыты" и "виртуальны"; все они везде доступны и ищутся динамически во время выполнения[1].

Тем не менее, на сегодняшний день в Python поддерживается понятие "корректировки" (т.е. расширения) имен для локализации некоторых имен в классах. Скорректированные имена иногда неправильно называют "закрытыми атрибутами", но в действительности они представляют собой всего лишь способ локализации имени в классе, который его создал — корректировка имен не предотвращает доступ к ним из кода за пределами класса. Данное средство предназначено главным образом для того, чтобы избежать конфликтов между пространствами имен в экземплярах, а не для ограничения доступа к именам в целом; следовательно, скорректированные имена лучше называть "псевдозакрытыми", чем "закрытыми".

Псевдозакрытые имена являются продвинутой и совершенно необязательной возможностью; вы вряд ли сочтете их крайне полезными, пока не начнете создавать универсальные инструменты или более крупные иерархии классов для применения в проектах, где принимает участие много программистов. На самом деле они не всегда используются, даже когда вероятно должны — чаще всего программисты на Python записывают внутренние имена с одиночным символом подчеркивания (наподобие `_X`). Это просто неформальное соглашение, которое уведомляет о том, что имя, как правило, не должно изменяться (для самого Python оно ничего не значит).

Однако поскольку вы можете видеть такой прием в коде других людей, то обязаны знать о нем, даже если сами его не применяете. И как только вы изучите его преимущества и контексты использования, то можете счесть данное средство более полезным в собственном коде, чем осознают некоторые программисты.

Обзор корректировки имен

Вот как работает корректировка имен: любые имена внутри оператора `class`, которые *начинаются* с двух символов подчеркивания, но не *заканчиваются* ими, автоматически расширяются, чтобы содержать впереди имя включающего класса. Например, имя вроде `__X` внутри класса `Spam` автоматически изменяется на `_Spam__X`: исходное

[1] Это способно несоразмерно пугать людей с опытом программирования на C++. В Python можно даже изменять или полностью удалять метод класса во время выполнения. С другой стороны, в реальных программах почти никто так не поступает. Как язык написания сценариев, Python больше заботится о разрешении, нежели об ограничении. Также вспомните из нашего обсуждения перегрузки операций в главе 30, что методы `__getattr__` и `__setattr__` могут применяться для эмуляции защиты, но на практике с такой целью они обычно не используются. Дополнительные сведения будут приведены при реализации более реалистичного декоратора защиты в главе 39.

имя снабжается префиксом в виде одиночного символа подчеркивания и именем включающего класса. Из-за того, что модифицированное имя содержит имя включающего класса, оно обычно уникально и не конфликтует с похожими именами, созданными другими классами в иерархии.

Корректировка имен происходит только для имен, появляющихся внутри кода оператора `class`, и затем только для имен, которые начинаются с двух символов подчеркивания. Тем не менее, она работает для *каждого* имени, предваренного двумя символами подчеркивания — атрибутов классов (в том числе имен методов) и атрибутов экземпляров, присвоенных в `self`. Скажем, в классе `Spam` метод по имени `__meth` после корректировки превращается в `_Spam__meth`, а ссылка на атрибут экземпляра `self.__X` видоизменяется на `self._Spam__X`.

Несмотря на корректировку, до тех пор, пока в классе везде, где есть ссылка на имя, применяется версия с двумя символами подчеркивания, все ссылки по-прежнему будут работать. Однако поскольку добавлять атрибуты к экземпляру могут сразу несколько классов, корректировка имен помогает избежать конфликтов — но чтобы увидеть, почему, нам необходимо перейти к конкретному примеру.

Для чего используются псевдозакрытые атрибуты?

Одна из основных проблем, которые призваны смягчить псевдозакрытые атрибуты, связана со способом хранения атрибутов экземпляров. В Python все атрибуты экземпляров оказываются в единственном объекте экземпляра в нижней части дерева классов и разделяются всеми функциями методов уровня класса, которым передается экземпляр. Такая модель отличается от модели, принятой в C++, где каждый класс получает собственное пространство для определяемых им данных-членов.

Внутри метода класса в Python всякий раз, когда метод присваивает значение какому-то атрибуту `self` (например, `self.`*`атрибут = значение`*), он изменяет или создает атрибут в экземпляре (вспомните, что поиск в иерархии наследования происходит только при ссылке, не в случае присваивания). Поскольку это справедливо, даже если множество классов в иерархии присваивают значение тому же самому атрибуту, возможны конфликты.

Скажем, пусть при реализации класса программист предполагает, что класс владеет атрибутом по имени `X` в экземпляре. В методах данного класса указанное имя устанавливается и позже извлекается:

```
class C1:
    def meth1(self): self.X = 88    # Я предполагаю, что атрибут X - мой
    def meth2(self): print(self.X)
```

Далее предположим, что еще один программист, работающий изолированно, делает то же самое допущение в другом классе:

```
class C2:
    def metha(self): self.X = 99    # Я тоже
    def methb(self): print(self.X)
```

Сами по себе оба класса функционируют. Проблема возникнет, если два класса когда-нибудь смешаются в одном дереве классов:

```
class C3(C1, C2): ...
I = C3()                            # В I есть только один атрибут X!
```

Теперь значение, которое каждый класс получает для выражения `self.X`, будет зависеть о того, какой класс выполнял присваивание последним. Из-за того, что все

присваивания self.X относятся к тому же самому одиночному экземпляру, существует только один атрибут X — I.X — вне зависимости от того, сколько классов применяют такое имя атрибута.

Это не проблема, если она ожидается, и так в действительности взаимодействуют классы — экземпляр является разделяемой памятью. Тем не менее, чтобы гарантировать принадлежность атрибута классу, который его использует, необходимо снабдить имя префиксом в виде двух символов подчеркивания везде, где оно применяется в классе, как показано в следующем файле pseudoprivate.py, работающем в Python 2.X/3.X:

```
class C1:
    def meth1(self): self.__X = 88       # Теперь я имею свой атрибут X
    def meth2(self): print(self.__X)     # Становится _C1__X в I
class C2:
    def metha(self): self.__X = 99       # Я тоже
    def methb(self): print(self.__X)     # Становится _C2__X в I

class C3(C1, C2): pass
I = C3()                                 # В I есть два имени X

I.meth1(); I.metha()
print(I.__dict__)
I.meth2(); I.methb()
```

При наличии таких префиксов перед добавлением к экземпляру атрибуты X будут расширены для включения имен своих классов. Если вы выполните вызов dir на I либо проинспектируете его словарь пространства имен после присваивания значений атрибутам, то увидите расширенные имена, _C1__X и _C2__X, а не X. Поскольку расширение делает имена более уникальными внутри экземпляра, разработчики классов могут вполне безопасно предполагать, что они по-настоящему владеют любыми именами, которые снабжают префиксами в форме двух символов подчеркивания:

```
% python pseudoprivate.py
{'_C2__X': 99, '_C1__X': 88}
88
99
```

Показанный трюк позволяет избежать потенциальных конфликтов имен в экземпляре, но учтите, что он не означает подлинную защиту. Зная имя включающего класса, вы по-прежнему можете получать доступ к любому из двух атрибутов везде, где имеется ссылка на экземпляр, с использованием полностью развернутого имени (например, I._C1__X = 77). Кроме того, имена все еще могут конфликтовать, когда неосведомленные программисты явно применяют расширенную схему именования (маловероятно, но возможно). С другой стороны, описанное средство делает менее вероятными *случайные* конфликты с именами какого-то класса.

Псевдозакрытые атрибуты также полезны в более крупных фреймворках либо инструментах, помогая избежать введения новых имен методов, которые могут неумышленно скрыть определения где-то в другом месте дерева классов, и снизить вероятность замещения внутренних методов именами, определяемыми ниже в дереве. Если метод предназначен для использования только внутри класса, который может смешиваться с другими классами, тогда префикс в виде двух символов подчеркивания фактически гарантирует, что снабженный им метод не будет служить препятствием другим именам в дереве, особенно в сценариях с множественным наследованием:

```
class Super:
   def method(self): ...                    # Действительный прикладной метод

class Tool:
   def __method(self): ...                  # Превращается в _Tool__method
   def other(self): self.__method()         # Использовать внутренний метод

class Sub1(Tool, Super): ...
   def actions(self): self.method() # Выполняется Super.method, как и ожидалось

class Sub2(Tool):
   def __init__(self): self.method = 99  # Не нарушает работу Tool.__method
```

Мы кратко упоминали множественное наследование в главе 26, и будем исследовать его более глубоко позже в текущей главе. Вспомните, что поиск в суперклассах производится в соответствии с их порядком слева направо в строках заголовка `class`. Здесь это означает, что класс `Sub1` отдает предпочтение атрибутам `Tool` перед атрибутами `Super`. Хотя в приведенном примере мы могли бы заставить Python выбирать методы прикладного класса первыми, изменив порядок указания суперклассов в заголовке класса `Sub1`, псевдозакрытые атрибуты решают проблему в целом. Псевдозакрытые имена также препятствуют случайному переопределению подклассами имен внутренних методов, как происходит в `Sub2`.

Опять-таки я должен отметить, что данная возможность имеет тенденцию применяться главным образом в более крупных проектах, где участвует много программистов, да и то лишь для избранных имен. Не поддавайтесь искушению излишне загромождать свой код; используйте эту возможность только для имен, которые действительно нуждаются в управлении со стороны единственного класса. Несмотря на полезность в ряде универсальных инструментов, основанных на классах, для более простых программ она, вероятно, будет излишеством.

За дополнительными примерами применения возможности именования `__X` обращайтесь к подмешиваемым классам в файле `lister.py`, представленном далее в главе, а также к обсуждению декораторов класса `Private` в главе 39.

Если вас заботит защита вообще, тогда можете еще раз просмотреть эмуляцию закрытых атрибутов экземпляров в разделе "Доступ к атрибутам: `__getattr__` и `__setattr__`" главы 30 и более полный декоратор класса `Private`, который будет построен с помощью делегирования в главе 39. Хотя в классах Python и возможна эмуляция подлинного управления доступом, на практике так поступают редко даже в крупных системах.

Методы являются объектами: связанные или несвязанные методы

Методы в целом и связанные методы в частности упрощают реализацию многих проектных целей на Python. Мы встречали связанные методы в главе 30 при изучении метода `__call__`. Как выяснится в приводимой здесь полной истории, они оказываются более общими и гибкими, чем можно было ожидать.

В главе 19 вы узнали, что функции можно обрабатывать как нормальные объекты. Методы тоже являются разновидностью объектов и могут использоваться таким же способом, как другие объекты — их разрешено присваивать именам, передавать в функции, сохранять в структурах данных и т.д. — и подобно простым функциям квалифицировать как объекты "первого класса". Однако поскольку доступ к методам класса возможен из экземпляра или класса, на самом деле в Python они поступают в двух вариантах.

***Объекты несвязанных методов (класса): без* `self`**

Доступ к функциональному атрибуту класса путем уточнения с помощью *класса* возвращает объект несвязанного метода. Чтобы вызвать метод, потребуется в первом аргументе явно указать объект экземпляра. В Python 3.X несвязанный метод представляет собой то же самое, что и простая функция, и может вызываться через имя класса; в Python 2.X он является отдельным типом и не может быть вызван без указания экземпляра.

Объекты связанных методов (экземпляра): пары* `self` *+ функция

Доступ к функциональному атрибуту класса путем уточнения с помощью *экземпляра* возвращает объект связанного метода. В объект связанного метода Python автоматически помещает экземпляр и функцию, так что для вызова метода передавать экземпляр не нужно.

Оба вида методов являются полноценными объектами; подобно строкам и числам их можно как угодно передавать в пределах программы. При запуске они оба также требуют указания экземпляра в своем первом аргументе (т.е. значения `self`). Вот почему мы должны были явно передавать экземпляр, когда вызывали методы суперкласса из методов подкласса в предшествующих примерах (включая файл `employees.py` в этой главе); формально такие вызовы попутно производят объекты *несвязанных* методов.

При вызове объекта *связанного* метода Python предоставляет экземпляр автоматически – экземпляр, применяемый для создания объекта связанного метода. Таким образом, объекты связанных методов обычно взаимозаменяемы с объектами простых функций, что делает их особенно удобными для интерфейсов, изначально ориентированных на функции (реалистичный сценарий использования в графических пользовательских интерфейсах описан во врезке "Что потребует внимания: обратные вызовы связанных методов" далее в главе).

В целях иллюстрации предположим, что мы определили следующий класс:

```
class Spam:
    def doit(self, message):
        print(message)
```

Теперь в нормальной операции мы создаем экземпляр и вызываем его метод, выводящий переданный аргумент:

```
object1 = Spam()
object1.doit('hello world')
```

Тем не менее, в действительности прямо перед круглыми скобками вызова метода генерируется объект *связанного* метода. Фактически мы можем извлечь связанный метод, не вызывая его. Подобно всем выражениям результатом выражения *`объект.имя`* будет объект. Ниже такое выражение возвращает объект связанного метода с упакованным экземпляром (`object1`) и функцией метода (`Spam.doit`). Мы можем присвоить эту пару связанного метода другому имени и затем вызвать его, как если бы оно было простой функцией:

```
object1 = Spam()
x = object1.doit        # Объект связанного метода: экземпляр + функция
x('hello world')        # Тот же эффект, что и object1.doit('...')
```

С другой стороны, если для того, чтобы добраться до `doit`, мы указываем класс, тогда получаем обратно объект *несвязанного* метода, который является просто ссылкой на

объект функции. Для вызова метода такого типа мы обязаны передать экземпляр как крайний слева аргумент — иначе он отсутствует в выражении, а метод его ожидает:

```
object1 = Spam()
t = Spam.doit        # Объект несвязанного метода (функция в Python 3.X: см. далее)
t(object1, 'howdy') # Передача экземпляра (если метод его ожидает в Python 3.X)
```

Более того, то же самое правило применяется внутри метода класса, если мы ссылаемся на атрибуты `self`, которые относятся к функциям в классе. Выражение `self.`*метод* представляет собой объект связанного метода, потому что `self` — объект экземпляра:

```
class Eggs:
    def m1(self, n):
        print(n)
    def m2(self):
        x = self.m1     # Еще один объект связанного метода
        x(42)           # Выглядит подобно простой функции

Eggs().m2()             # Выводит 42
```

В большинстве случаев вы вызываете методы немедленно после их извлечения посредством уточнения атрибутов, поэтому не всегда замечаете попутно сгенерированные объекты методов. Но когда вы начнете писать код, который вызывает объекты обобщенным образом, вам придется позаботиться о специальной трактовке несвязанных методов — они обычно требуют передачи явного объекта экземпляра.

Дополнительное исключение из данного правила ищите в обсуждении *статических методов и методов класса* в следующей главе и при кратком упоминании в следующем разделе. Подобно связанным методам статические методы могут маскироваться под базовые функции, т.к. при вызове они не ожидают передачи экземпляров. Говоря формально, Python поддерживает три вида методов уровня класса (методы экземпляров, статические методы и методы класса), а Python 3.X также позволяет присутствовать в классах простым функциям. Методы метаклассов, рассматриваемые в главе 40, тоже являются отдельными, но по существу они представляют собой методы класса с меньшей областью видимости.

Несвязанные методы являются функциями в Python 3.X

В Python 3.X из языка было исключено понятие *несвязанных методов*. То, что здесь мы описывали как несвязанный метод, в Python 3.X трактуется как *простая функция*. В большинстве случаев вашему коду это безразлично; оба способа предусматривают передачу экземпляра в первом аргументе при вызове метода через экземпляр.

Однако это может повлиять на программы, которые выполняют явную проверку типов — если вы выводите тип метода уровня класса без экземпляра, то получите "unbound method" ("несвязанный метод") в Python 2.X и "function" ("функция") в Python 3.X.

Кроме того, в Python 3.X допускается вызывать метод без экземпляра при условии, что метод его не ожидает, и метод вызывается только через *класс* и никогда через экземпляр. То есть Python 3.X будет передавать экземпляр методам только для вызовов через экземпляр. При вызове через класс передавать экземпляр вручную понадобится только в случае, если метод его ожидает:

```
C:\code> c:\python37\python
>>> class Selfless:
        def __init__(self, data):
            self.data = data
        def selfless(arg1, arg2):        # Простая функция в Python 3.X
            return arg1 + arg2
        def normal(self, arg1, arg2):    # При вызове ожидается экземпляр
            return self.data + arg1 + arg2

>>> X = Selfless(2)
>>> X.normal(3, 4)              # Экземпляр передается self автоматически: 2+(3+4)
9
>>> Selfless.normal(X, 3, 4)    # Метод ожидает self: передать вручную
9
>>> Selfless.selfless(3, 4)     # Без передачи экземпляра: в Python 3.X
                                #   работает, в Python 2.X - нет!
7
```

Последний тест в Python 2.X потерпит неудачу, потому что несвязанные методы по умолчанию требуют передачи экземпляра; в Python 3.X он работает из-за того, что такие методы трактуются как простые функции, не нуждающиеся в экземпляре. Несмотря на то что в Python 3.X перестают отлавливаться некоторые потенциальные ошибки (что, если программист забыл передать экземпляр?), появляется возможность использовать методы класса как простые функции до тех пор, пока им не передается аргумент экземпляра `self` и они не рассчитывают на него.

Тем не менее, следующие два вызова по-прежнему терпят неудачу в Python 3.X и 2.X. Первый (вызов через экземпляр) автоматически передает экземпляр методу, который его не ожидает, в то время как второй (вызов через класс) не передает экземпляр методу, который его ожидает (сообщения об ошибках получены в версии Python 3.7):

```
>>> X.selfless(3, 4)
TypeError: selfless() takes 2 positional arguments but 3 were given
Ошибка типа: selfless() принимает 2 позиционных аргумента, но было предоставлено 3

>>> Selfless.normal(3, 4)
TypeError: normal() missing 1 required positional argument: 'arg2'
Ошибка типа: в normal() отсутствует 1 обязательный позиционный аргумент: arg2
```

Из-за такого изменения встроенная функция `staticmethod` и декоратор, описанный в следующей главе, не нужны в Python 3.X для методов без аргумента `self`, которые вызываются только через имя *класса* и никогда через экземпляр — такие методы запускаются как простые функции, не получая аргумент экземпляра. В Python 2.X подобные вызовы приводят к ошибкам, если только экземпляр не передан вручную или метод не помечен как статический (статические методы более подробно обсуждаются в следующей главе).

Об отличиях в поведении Python 3.X знать важно, но в любом случае связанные методы обычно важнее с практической точки зрения. Поскольку они объединяют вместе экземпляр и функцию в единый объект, их обобщенно можно трактовать как вызываемые объекты. В следующем разделе все сказанное демонстрируется в коде.

Более наглядную иллюстрацию обработки несвязанных методов в Python 3.X и 2.X ищите в примере `lister.py` позже в настоящей главе. Его классы выводят значения методов, извлеченные из экземпляров и классов, в обеих линейках Python — как несвязанных методов в Python 2.X и как простых функций в Python 3.X. Также обратите внимание, что изменение присуще самой линейке Python 3.X, а не обязательной в ней модели классов нового стиля.

Связанные методы и другие вызываемые объекты

Как упоминалось ранее, связанные методы могут обрабатываться как обобщенные объекты подобно простым функциям – их можно произвольно передавать в рамках программы. Кроме того, поскольку связанные методы объединяют функцию и экземпляр в единый пакет, они могут трактоваться как любой другой вызываемый объект и не требовать специального синтаксиса при вызове. Например, в следующем коде четыре объекта связанных методов сохраняются в списке и позже вызываются с помощью нормальных выражений вызовов:

```
>>> class Number:
        def __init__(self, base):
            self.base = base
        def double(self):
            return self.base * 2
        def triple(self):
            return self.base * 3

>>> x = Number(2)                          # Объекты экземпляров класса
>>> y = Number(3)                          # Состояние + методы
>>> z = Number(4)
>>> x.double()                             # Нормальные прямые вызовы
4

>>> acts = [x.double, y.double, y.triple, z.double] # Список связанных методов
>>> for act in acts:                       # Вызовы откладываются
        print(act())                       # Вызов, как если бы это были функции

4
6
9
8
```

Подобно простым функциям объекты связанных методов имеют собственную информацию для интроспекции, включая атрибуты, которые предоставляют доступ к образующим пару объекту экземпляра и функции метода. Вызов связанного метода просто направляется этой паре:

```
>>> bound = x.double
>>> bound.__self__, bound.__func__
(<__main__.Number object at 0x...etc...>, <function Number.double at 0x...etc...>)
>>> bound.__self__.base
2
>>> bound()     # Вызывается bound.__func__(bound.__self__, ...)
4
```

Другие вызываемые объекты

На самом деле связанные методы являются всего лишь одним из нескольких типов вызываемых объектов в Python. Как демонстрируется ниже, простые функции, определенные посредством `def` или `lambda`, экземпляры, которые наследуют `__call__`, и связанные методы экземпляров могут обрабатываться и вызываться одинаково:

```
>>> def square(arg):
        return arg ** 2                    # Простые функции (def или lambda)

>>> class Sum:
        def __init__(self, val):           # Вызываемые экземпляры
            self.val = val
```

```
        def __call__(self, arg):
            return self.val + arg

>>> class Product:
        def __init__(self, val):                # Связанные методы
            self.val = val
        def method(self, arg):
            return self.val * arg

>>> sobject = Sum(2)
>>> pobject = Product(3)
>>> actions = [square, sobject, pobject.method] # Функция, экземпляр, метод

>>> for act in actions:                         # Все три вызываются одинаково
        print(act(5))                           # Вызов любого вызываемого
                                                #   объекта с одним аргументом

25
7
15
>>> actions[-1](5)                   # Индексирование, включение, отображение
15
>>> [act(5) for act in actions]
[25, 7, 15]
>>> list(map(lambda act: act(5), actions))
[25, 7, 15]
```

Говоря формально, классы тоже относятся к категории вызываемых объектов, но мы обычно вызываем их для создания экземпляров, а не для выполнения фактической работы – одиночное действие лучше реализовывать как простую функцию, чем класс с конструктором, но показанный далее класс служит иллюстрацией своей вызываемой природы:

```
>>> class Negate:
      def __init__(self, val): # Классы также являются вызываемыми объектами
          self.val = -val      # Но вызываются для создания объектов,
                               #   не для выполнения работы
      def __repr__(self):      # Формат вывода экземпляров
          return str(self.val)

>>> actions = [square, sobject, pobject.method, Negate]  # Вызвать также и класс
>>> for act in actions:
        print(act(5))

25
7
15
-5
>>> [act(5) for act in actions]          # Выполняется __repr__, а не __str__!
[25, 7, 15, -5]

>>> table = {act(5): act for act in actions} # Включение словаря из Python 3.X/2.7
>>> for (key, value) in table.items():
      print('{0:2} => {1}'.format(key, value)) # str.format из Python 2.6+/3.X

25 => <function square at 0x0000000002987400>
15 => <bound method Product.method of <__main__.Product object at ...etc...>>
-5 => <class '__main__.Negate'>
 7 => <__main__.Sum object at 0x000000000298BE48>
```

Как видите, связанные методы и в целом модель вызываемых объектов Python входят в состав многочисленных способов, заложенных в проектное решение Python, которые делают его невероятно гибким языком.

Теперь вы должны понимать объектную модель методов. Другие примеры связанных методов в работе приведены ниже во врезке "Что потребует внимания: обратные вызовы связанных методов", а также в разделе "Выражения вызовов: `__call__`" предыдущей главы.

Что потребует внимания: обратные вызовы связанных методов

Поскольку связанные методы автоматически образуют пару из экземпляра и функции метода класса, их можно применять везде, где ожидается простая функция. Одним из наиболее распространенных мест, где вы встретите воплощение этой идеи, является код регистрации методов как обработчиков событий в интерфейсе tkinter (Tkinter в Python 2.X), с которым мы уже сталкивались ранее. Вот простой случай:

```
def handler():
    ...использовать для хранения состояния глобальные переменные или
области видимости замыканий...
...
widget = Button(text='spam', command=handler)
```

Чтобы зарегистрировать обработчик для событий щелчков на кнопке, мы обычно передаем в ключевом аргументе `command` вызываемый объект, не принимающий аргументов. Здесь подходят имена функций (и `lambda`), а потому методы уровня класса, хотя они должны быть связанными методами, если при вызове ожидают указания экземпляра:

```
class MyGui:
    def handler(self):
        ...использовать для хранения состояния self.attr...
    def makewidgets(self):
        b = Button(text='spam', command=self.handler)
```

Обработчиком событий является `self.handler` — объект связанного метода, который запоминает `self` и `MyGui.handler`. Поскольку `self` будет ссылаться на исходный экземпляр, когда `handler` позже вызывается при поступлении событий, метод будет иметь доступ к атрибутам экземпляра, которые способны предохранять состояние между событиями, а также к методам уровня класса. В случае простых функций состояние взамен должно сохраняться в глобальных переменных или областях видимости объемлющих функций.

Еще один способ обеспечения совместимости классов с API-интерфейсами, основанными на функциях, был описан при обсуждении метода перегрузки операций `__call__` в главе 30, а другой инструмент, часто используемый при обратных вызовах, рассматривался при исследовании `lambda` в главе 19 первого тома. Как там отмечалось, обычно вам не нужно помещать связанный метод внутрь `lambda`; связанный метод в предыдущем примере уже откладывает вызов (обратите внимание на отсутствие круглых скобок, инициирующих вызов), поэтому добавлять `lambda` было бы бессмысленно!

Классы являются объектами: обобщенные фабрики объектов

Временами проектные решения на основе классов требуют создания объектов в ответ на условия, которые невозможно предсказать на стадии написания кода программы. Паттерн проектирования "Фабрика" делает возможным такой отложенный подход. Во многом благодаря гибкости Python фабрики могут принимать многочисленные формы, ряд которых вообще не кажутся особыми.

Из-за того, что классы также являются объектами "первого класса", их легко передавать в рамках программы, сохранять в структурах данных и т.д. Вы также можете передавать классы функциями, которые генерируют произвольные виды объектов; в кругах объектно-ориентированного проектирования такие функции иногда называются *фабриками*. Фабрики могут оказаться крупным делом в строго типизированных языках вроде C++, но довольно просты в реализации на Python.

Скажем, синтаксис вызова, представленный в главе 18 первого тома, позволяет вызывать любой класс с любым количеством позиционных или ключевых аргументов конструктора за один шаг для создания экземпляра любого вида [2]:

```
def factory(aClass, *pargs, **kargs): # Кортеж или словарь с переменным
                                      # количеством аргументов
    return aClass(*pargs, **kargs)   # Вызывает aClass (или apply в Python 2.X)

class Spam:
    def doit(self, message):
        print(message)

class Person:
    def __init__(self, name, job=None):
        self.name = name
        self.job = job

object1 = factory(Spam)                        # Создать объект Spam
object2 = factory(Person, "Arthur", "King")    # Создать объект Person
object3 = factory(Person, name='Brian')     # То же самое, с ключевым аргу-
                                            # ментом и стандартным значением
```

В коде мы определяем функцию генератора объектов по имени `factory`. Она ожидает передачи объекта класса (подойдет любой класс) наряду с одним и более аргументов для конструктора класса. Для вызова функции и возвращения экземпляра применяется специальный синтаксис с переменным количеством аргументов.

В оставшемся коде примера просто определяются два класса и генерируются экземпляры обоих классов за счет их передачи функции `factory`. И это единственная фабричная функция, которую вам когда-либо придется писать на Python; она работает для любого класса и любых аргументов конструкторов. Ниже функция `factory` демонстрируется в действии (файл `factory.py`):

---

[2] В действительности этот синтаксис дает возможность вызывать любой вызываемый объект, включая функции, классы и методы. Следовательно, функция `factory` здесь может также запускать любой вызываемый объект, а не только класс (несмотря на имя аргумента). Кроме того, как было показано в главе 18 первого тома, в Python 2.X имеется альтернатива `aClass(*pargs, **kargs)`: встроенный вызов `apply(aClass, pargs, kargs)`, который был удален в Python 3.X из-за своей избыточности и ограничений.

```
>>> object1.doit(99)
99
>>> object2.name, object2.job
('Arthur', 'King')
>>> object3.name, object3.job
('Brian', None)
```

К настоящему времени вы должны знать, что абсолютно все в Python является объектом "первого класса" — в том числе классы, которые обычно представляют собой лишь входные данные для компилятора в языках, подобных C++. Передавать их таким способом вполне естественно. Однако как упоминалось в начале текущей части книги, задачи ООП в Python решаются только с помощью объектов, *полученных* из классов.

Для чего используются фабрики?

Так в чем же польза от функции `factory` (помимо повода проиллюстрировать в книге объекты классов)? К сожалению, трудно показать приложения паттерна проектирования "Фабрика" без приведения кода большего объема, чем для этого есть место. Тем не менее, в целом такая фабрика способна сделать возможной изоляцию кода от деталей динамически конфигурируемого создания объектов.

Например, вспомните пример функции `processor`, абстрактно представленной в главе 26, и затем снова в качестве примера композиции ранее в этой главе. Она принимает объекты `reader` и `writer` для обработки произвольных потоков данных. Первоначальной версии функции `processor` вручную передавались экземпляры специализированных классов вроде `FileWriter` и `SocketReader` с целью настройки обрабатываемых потоков данных; позже мы передавали жестко закодированные объекты файла, потока и форматера. В более динамическом сценарии для настройки потоков данных могли бы применяться внешние механизмы, такие как конфигурационные файлы или графические пользовательские интерфейсы.

В динамическом мире подобного рода может отсутствовать возможность жесткого кодирования в сценариях процедуры для создания объектов интерфейса к потокам данных и взамен их придется создавать во время выполнения в соответствии с содержимым конфигурационного файла.

Такой файл может просто задавать строковое имя класса потока данных, подлежащего импортированию из модуля, плюс необязательный аргумент для вызова конструктора. Здесь могут пригодиться функции или код в стиле фабрик, потому что он позволяет извлекать и передавать классы, код которых не реализован заблаговременно в программе. На самом деле классы могут даже вообще не существовать в момент, когда пишется код:

```
classname = ...извлечь из конфигурационного файла и произвести разбор...
classarg = ...извлечь из конфигурационного файла и произвести разбор...
import streamtypes                          # Настраиваемый код
aclass = getattr(streamtypes, classname)    # Извлечь из модуля
reader = factory(aclass, classarg)          # Или aclass(classarg)
processor(reader, ...)
```

Здесь снова используется встроенная функция `getattr` для извлечения атрибута модуля по строковому имени (она похожа на выражение *объект*.*атрибут*, но *атрибут* является строкой). Поскольку в приведенном фрагменте кода предполагается наличие у конструктора единственного аргумента, строго говоря, он не нуждается в функции `factory` — мы могли бы создавать экземпляр с помощью `aclass(classarg)`. Однако фабричная функция может оказаться более полезной при наличии неизвестных списков аргументов, а общий паттерн проектирования "Фабрика" способен повысить гибкость кода.

Множественное наследование: "подмешиваемые" классы

Наш последний паттерн проектирования является одним из самых полезных и послужит предметом для построения более реалистичного примера, чтобы завершить настоящую главу и подготовить почву для дальнейших исследований. В качестве бонуса код, который мы здесь напишем, может стать удобным инструментом.

Многие проектные решения, основанные на классах, требуют объединения разрозненных наборов методов. Как уже было показано, в строке заголовка оператора `class` в круглых скобках можно указывать более одного суперкласса. Поступая так, мы задействуем *множественное наследование* — класс и его экземпляры наследуют имена из *всех* перечисленных суперклассов.

Для нахождения атрибута процедура поиска в иерархии наследования Python обходит все суперклассы, указанные в заголовке оператора `class`, слева направо до тех пор, пока не обнаружит соответствие. Формально из-за того, что любой из суперклассов может иметь собственные суперклассы, такой поиск может быть чуть сложнее для более крупных деревьев классов.

- В классических классах (стандарт вплоть до Python 3.0) поиск атрибутов во всех случаях продолжается сначала в глубину на всем пути к вершине дерева наследования и затем слева направо. Такой порядок обычно называется DFLR (Depth-First, Left-to-Right — сначала в глубину, слева направо).
- В классах нового стиля (необязательные в Python 2.X и стандарт в Python 3.X) поиск атрибутов обычно происходит, как было ранее, но в ромбовидных схемах продолжается по уровням дерева до перехода вверх, т.е. больше в манере сначала в ширину. Такой порядок обычно называется MRO нового стиля (Method Resolution Order — порядок распознавания методов), хотя он применяется для всех атрибутов, а не только для методов.

Второе из описанных правил поиска исчерпывающе объясняется при обсуждении классов нового стиля в следующей главе. Несмотря на то что без кода следующей главы *ромбовидные* схемы понять трудно (и создавать их самостоятельно доведется редко), они образуются, когда множество классов в дереве используют общий суперкласс; порядок поиска нового стиля спроектирован так, чтобы посещать разделяемый суперкласс только один раз и после всех его подклассов. Тем не менее, в любой из двух моделей, когда класс имеет множество суперклассов, поиск в них производится слева направо согласно порядку указания суперклассов в строках заголовка оператора `class`.

В целом множественное наследование хорошо подходит для моделирования объектов, которые принадлежат более чем одному набору. Скажем, человек может быть инженером, писателем, музыкантом и т.д., а потому наследовать свойства от всех наборов подобного рода. Благодаря множественному наследованию объекты получают объединение всех линий поведения из всех своих суперклассов. Вскоре мы увидим, что множественное наследование также позволяет классам функционировать в качестве универсальных пакетов смешиваемых атрибутов.

Хотя множественное наследование — полезный паттерн проектирования, его главный недостаток в том, что оно может привести к *конфликту*, когда то же самое имя метода (или другого атрибута) определено сразу в нескольких суперклассах.

Возникший конфликт разрешается либо автоматически за счет порядка поиска в иерархии наследования, либо вручную в коде.

- ***Стандартный способ.*** По умолчанию процедура поиска в иерархии наследования выбирает первое найденное вхождение атрибута, когда на атрибут производится ссылка обычным образом, например, `self.method()`. В таком режиме Python выбирает самый нижний и крайний слева атрибут в классических классах и при неромбовидных схемах во всех классах; в классах нового стиля при ромбовидных схемах может быть выбран вариант справа или выше.
- ***Явный способ.*** В некоторых моделях на основе классов иногда необходимо выбирать атрибут явно, ссылаясь на него через имя класса, скажем, `superclass.method(self)`. Ваш код разрешает конфликт и переопределяет стандартный способ поиска – чтобы выбрать вариант справа или выше принятого по умолчанию при поиске в иерархии наследования.

Проблема возникает, только когда *то же самое имя* появляется во множестве суперклассов, и вы не хотите использовать первое унаследованное. Поскольку в типичном коде на Python она не настолько распространена, как может показаться, мы отложим исследование деталей до следующей главы, где рассматриваются классы нового стиля вместе с их инструментами MRO и `super`, а также возвратимся к ней при анализе затруднений в конце главы. Однако сначала мы продемонстрируем практический сценарий применения для инструментов, основанных на множественном наследовании.

Реализация подмешиваемых классов отображения

Возможно, множественное наследование чаще всего используется для "подмешивания" универсальных методов из суперклассов. Такие суперклассы обычно называются *подмешиваемыми классами* – они предоставляют методы, которые добавляются к прикладным классам через наследование. В некотором смысле подмешиваемые классы похожи на модули: они предлагают пакеты методов для применения в своих клиентских подклассах. Тем не менее, в отличие от простых функций методы в подмешиваемых классах также могут принимать участие в иерархиях наследования и иметь доступ к экземпляру `self` для использования информации о состоянии и других методов в своих деревьях.

Например, как мы видели, стандартный способ, которым Python выводит объект экземпляра класса, не особенно практичен:

```
>>> class Spam:
        def __init__(self):          # Метод __repr__ или __str__ отсутствует
            self.data1 = "food"

>>> X = Spam()
>>> print(X)          # Стандартный способ: имя класса + адрес (идентификатор)
<__main__.Spam object at 0x00000000029CA908>     # То же самое в Python 2.X,
но всегда instance
```

В учебном примере из главы 28 и при обсуждении перегрузки операций в главе 30 было показано, что вы сами можете предоставлять метод `__str__` или `__repr__` для реализации специального строкового представления. Но вместо написания кода одного из них в каждом классе, который желательно выводить, почему бы не реализовать данный метод один раз в классе универсального инструмента и наследовать его во всех своих классах?

Именно для этого предназначены подмешиваемые классы. Однократное определение метода отображения в подмешиваемом суперклассе позволяет его многократно применять везде, где требуется специальный формат вывода — даже в классах, которые уже могут иметь другой суперкласс. Мы уже видели инструменты, выполняющие связанную работу.

- Класс `AttrDisplay` из главы 28 форматировал атрибуты экземпляров в обобщенном методе `__repr__`, но не поднимался по дереву классов и задействовал только режим одиночного наследования.
- Модуль `classtree.py` из главы 29 определял функции для подъема по деревьям классов и их схематического изображения, но он попутно не отображал атрибуты объектов и не был спроектирован как наследуемый класс.

Здесь мы собираемся возвратиться к методикам указанных примеров и расширить их, чтобы реализовать набор из трех подмешиваемых классов, которые будут служить обобщенными инструментами отображения для списков атрибутов экземпляров, унаследованных атрибутов и атрибутов всех объектов в дереве классов. Мы будем также использовать созданные инструменты в режиме множественного наследования и введем в действие кодовые методики, которые сделают классы лучше подходящими для применения в качестве обобщенных инструментов.

В отличие от главы 28 мы построим реализацию на основе метода `__str__`, а не `__repr__`. Отчасти это проблема стиля, а роль инструментов ограничивается `print` и `str`, но разрабатываемые отображения будут достаточно обогащенными, чтобы считаться более дружественными к пользователю, чем представление как в коде. Такая политика также оставляет клиентским классам возможность реализации альтернативного низкоуровневого отображения посредством `__repr__` при эхо-выводе в интерактивной подсказке и для вложенных появлений. Использование `__repr__` по-прежнему допускает альтернативную версию `__str__`, но *природа* отображений, которые мы реализуем более строго, предполагает применение `__str__`. Отличия между `__str__` и `__repr__` обсуждались в главе 30.

Вывод списка атрибутов экземпляра с помощью `__dict__`

Давайте начнем с простого случая — вывод списка атрибутов, присоединенных к экземпляру. В показанном далее коде, находящемся в файле `listinstance.py`, определяется подмешиваемый класс по имени `ListInstance`, который перегружает метод `__str__` для всех классов, содержащих его в строках заголовков своих операторов `class`. Из-за реализации в виде класса `ListInstance` является обобщенным инструментом, чью логику форматирования можно использовать для экземпляров любого клиентского подкласса:

```
#!python
# Файл listinstance.py (Python 2.X + 3.X)

class ListInstance:
    """
    Подмешиваемый класс, который предоставляет форматированную функцию
    print() или str() для экземпляров через наследование реализованного
    в нем метода __str__; отображает только атрибуты экземпляра; self
    является экземпляром самого нижнего класса;
    имена __X предотвращают конфликты с атрибутами клиента
    """
    def __attrnames(self):
        result = ''
```

```
        for attr in sorted(self.__dict__):
            result += '\t%s=%s\n' % (attr, self.__dict__[attr])
        return result

    def __str__(self):
        return '<Instance of %s, address %s:\n%s>' % (
                            self.__class__.__name__,       # Имя класса
                            id(self),                      # Адрес
                            self.__attrnames())            # Список имя=значение

if __name__ == '__main__':
    import testmixin
    testmixin.tester(ListInstance)
```

Весь приводимый в разделе код выполняется в Python 2.X и 3.X. Есть одно замечание: данный код демонстрирует классическую схему включения, и вы могли бы уменьшить его объем за счет более лаконичной реализации метода `__attrnames` посредством *генераторного выражения*, запускаемого строковым методом `join`, но результат оказался бы менее ясным — такие выражения обычно должны подталкивать к поиску более простых альтернатив:

```
def __attrnames(self):
    return ''.join('\t%s=%s\n' % (attr, self.__dict__ [attr])
        for attr in sorted(self.__dict__))
```

Для извлечения имени класса и атрибутов экземпляра в `ListInstance` применяются ранее исследованные приемы.

- Каждый экземпляр имеет встроенный атрибут `__class__`, ссылающийся на класс, из которого он был создан, а каждый класс содержит атрибут `__name__`, ссылающийся на имя в заголовке, так что выражение `self.__class__.__name__` извлекает имя класса экземпляра.
- Класс `ListInstance` выполняет большую часть своей работы, просто просматривая словарь атрибутов экземпляра (вспомните, что он экспортируется в `__dict__`) для формирования строки с именами и значениями всех атрибутов экземпляра. Ключи словаря сортируются, чтобы обойти любые отличия в упорядочивании между выпусками Python.

В этих отношениях класс `ListInstance` похож на реализацию отображения атрибутов в главе 28; фактически его можно считать лишь вариацией на данную тему. Однако наш класс использует две дополнительные методики.

- Он отображает адрес экземпляра в памяти, вызывая встроенную функцию `id`, которая возвращает адрес любого объекта (по определению уникальный идентификатор объекта, который будет полезен при дальнейшей модификации кода).
- Он применяет схему псевдозакрытого именования для своего рабочего метода: `__attrnames`. Как объяснялось ранее в главе, любое имя подобного рода Python автоматически локализует во включающем его классе, дополняя имя атрибута именем класса (в данном случае оно превращается в `_ListInstance__attrnames`). Это остается справедливым для атрибутов класса (вроде методов) и атрибутов экземпляра, присоединенных к `self`. Как отмечалось в пробной версии в главе 28, такое поведение полезно в универсальных инструментах, поскольку оно гарантирует, что их имена не будут конфликтовать с любыми именами, используемыми в клиентских подклассах.

Из-за того, что ListInstance определяет метод перегрузки операции __str__, создаваемые из этого класса экземпляры автоматически отображают свои атрибуты при выводе, давая чуть больше информации, чем простой адрес. Ниже показана работа класса в режиме одиночного наследования, когда он подмешивается к классу из предыдущего раздела (код выполняется в Python 3.X и 2.X одинаково, хотя repr из Python 2.X по умолчанию отображает метку instance, а не object):

```
>>> from listinstance import ListInstance
>>> class Spam(ListInstance):                # Наследует метод __str__
        def __init__(self):
            self.data1 = 'food'

>>> x = Spam()
>>> print(x)                                 # print() и str() запускают __str__
<Instance of Spam, address 43034496:
        data1=food
>
```

Вы можете также извлекать и сохранять выходной список в виде строки, не выводя его посредством str, а эхо-вывод интерактивной подсказки по-прежнему применяет стандартный формат, потому что возможность реализации метода __repr__ мы оставили клиентам как вариант:

```
>>> display = str(x) # Вывести строку для интерпретации управляющих символов
>>> display
'<Instance of Spam, address 43034496:\n\tdata1=food\n>'

>>> x                    # Метод __repr__ по-прежнему использует стандартный формат
<__main__.Spam object at 0x000000000290A780>
```

Класс ListInstance пригоден для любого создаваемого класса — даже классов, которые уже имеют один и более суперклассов. Именно здесь пригодится *множественное наследование*: за счет добавления ListInstance к перечню суперклассов в строках заголовка оператора class (т.е. подмешивая класс ListInstance) вы получаете его метод __str__ "бесплатно" наряду с тем, что наследуете из существующих суперклассов. В файле testmixin0.py приведен пробный тестовый сценарий:

```
# Файл testmixin0.py
from listinstance import ListInstance   # Получить класс инструмента для
вывода списка атрибутов

class Super:
    def __init__(self):                 # Метод __init__ суперкласса
        self.data1 = 'spam'             # Создать атрибуты экземпляра
    def ham(self):
        pass

class Sub(Super, ListInstance):         # Подмешивание ham и __str__
    def __init__(self):                 # Классы, выводящие списки атрибутов,
                                        #   имеют доступ к self
        Super.__init__(self)
        self.data2 = 'eggs'             # Дополнительные атрибуты экземпляра
        self.data3 = 42
    def spam(self):                     # Определить здесь еще один метод
        pass

if __name__ == '__main__':
    X = Sub()
    print(X)                            # Выполняется подмешанный метод __str__
```

Класс Sub наследует имена из Super и ListInstance; он содержит собственные имена и имена из обоих суперклассов. Когда вы создаете экземпляр Sub и выводите его, то автоматически получаете специальное представление, подмешанное из ListInstance (в данном случае вывод сценария одинаков в Python 3.X и 2.X, исключая адреса объектов, которые вполне естественно могут варьироваться от процесса к процессу):

```
c:\code> python testmixin0.py
<Instance of Sub, address 44304144:
        data1=spam
    data2=eggs
    data3=42
>
```

Тестовый сценарий testmixin0 работает, но имя тестируемого класса в нем жестко закодировано, что затрудняет экспериментирование с альтернативными версиями – чем мы вскоре займемся. Для обеспечения более высокой гибкости мы можем позаимствовать код из примера с перезагрузкой модулей в главе 25 первого тома и передавать объект, подлежащий тестированию, как иллюстрируется в показанном ниже усовершенствованном тестовом сценарии testmixin, который фактически используется в коде самотестирования всех модулей с классами вывода списков. В данном контексте передаваемый инструменту тестирования объект является подмешиваемым *классом*, а не функцией, но принцип похож: в Python абсолютно все квалифицируется как объект "первого класса":

```
#!python
# Файл testmixin.py (Python 2.X + 3.X)
"""
Обобщенный инструмент тестирования подмешиваемых классов вывода списков:
он похож на средство транзитивной перезагрузки модулей из главы 25 первого
тома, но ему передается объект класса (не функции), а в testByNames
добавлена загрузка модуля и класса по строковым именам в соответствии с
паттерном проектирования 'Фабрика'.
"""
import importlib

def tester(listerclass, sept=False):

    class Super:
        def __init__(self):            # Метод __init__ суперкласса
            self.data1 = 'spam'        # Создать атрибуты экземпляра
        def ham(self):
            pass

    class Sub(Super, listerclass):  # Подмешивание ham и __str__
        def __init__(self):            # Классы, выводящие списки атрибутов,
                                       #   имеют доступ к self
            Super.__init__(self)
            self.data2 = 'eggs'        # Дополнительные атрибуты экземпляра
            self.data3 = 42
        def spam(self):                # Определить здесь еще один метод
            pass

    instance = Sub()                   # Возвратить экземпляр с помощью __str__
                                       #   класса, выводящего список
    print(instance)                    # Выполняется подмешанный метод __str__
                                       #   (или через str(x))
    if sept: print('-' * 80)
```

```
def testByNames(modname, classname, sept=False):
    modobject = importlib.import_module(modname)      # Импортировать по
                                                      #   строковым именам
    listerclass = getattr(modobject, classname)       # Извлечь атрибуты по
                                                      #   строковым именам
    tester(listerclass, sept)

if __name__ == '__main__':
    testByNames('listinstance', 'ListInstance', True) # Протестировать все
                                                      #   три класса
    testByNames('listinherited', 'ListInherited', True)
    testByNames('listtree', 'ListTree', False)
```

Одновременно в сценарий также добавлена возможность указывать тестируемый модуль и класс по строковым именам, которая задействована в его коде самотестирования – приложение механики описанного ранее паттерна проектирования "Фабрика". Ниже новый сценарий демонстрируется в работе, запускаемый модулем с классом вывода списка, который импортирует его для тестирования собственного класса (снова с одинаковыми результатами в Python 2.X и 3.X). Можно также запустить сам тестовый сценарий, но в этом режиме тестируются два варианта класса вывода списка, которые мы пока еще не видели (и не реализовывали!):

```
c:\code> python listinstance.py
<Instance of Sub, address 43256968:
        data1=spam
        data2=eggs
        data3=42
>

c:\code> python testmixin.py
<Instance of Sub, address 43977584:
        data1=spam
        data2=eggs
        data3=42
>
...и результаты тестов двух других классов вывода списков, которые еще
предстоит создать...
```

Реализованный до сих пор класс `ListInstance` работает в любом классе, куда он подмешивается, потому что `self` ссылается на экземпляр подкласса, в который помещается `ListInstance`, каким бы он ни был. Опять-таки в известной мере подмешиваемые классы представляют собой классовый эквивалент модулей – пакеты методов, полезных в разнообразных клиентах. Скажем, ниже показано, как `ListInstance` работает в режиме одиночного наследования с экземплярами другого класса, загруженного с помощью `import`, и отображает атрибуты, значения которым присваиваются за пределами класса:

```
>>> import listinstance
>>> class C(listinstance.ListInstance): pass

>>> x = C()
>>> x.a, x.b, x.c = 1, 2, 3
>>> print(x)
<Instance of C, address 43230824:
        a=1
        b=2
        c=3
>
```

Помимо обеспечиваемой подмешиваемыми классами полезности они оптимизируют сопровождение кода, как и все классы. Например, если позже вы решите расширить метод __str__ класса ListInstance путем добавления вывода всех атрибутов класса, унаследованных экземпляром, то безопасно можете сделать это; поскольку метод __str__ наследуемый, его изменение автоматически обновляет отображение каждого подкласса, который импортирует и подмешивает класс ListInstance. И так как теперь официально "позже" уже наступило, давайте перейдем к следующему разделу и выясним, как может выглядеть такое расширение.

Вывод списка унаследованных атрибутов с помощью dir

В том виде, как есть, наш подмешиваемый класс ListerInstance отображает только атрибуты экземпляра (т.е. имена, присоединенные к самому объекту экземпляра). Тем не менее, класс легко расширить для отображения всех атрибутов, доступных из экземпляра – собственных и унаследованных из его классов. Трюк предусматривает применение встроенной функции dir вместо просмотра словаря __dict__; словарь хранит только атрибуты экземпляра, но функция также собирает все унаследованные атрибуты в Python 2.2 и последующих версиях.

Описанная схема реализована в приведенной далее модификации; она помещена в собственный модуль для облегчения тестирования, но если бы существующие клиенты взамен использовали данную версию, тогда они получили бы новое отображение автоматически (и вспомните из главы 25 первого тома, что конструкция as оператора import позволяет назначать новой версии ранее применяемое имя):

```
#!python
# Файл listinherited.py (Python 2.X + 3.X)

class ListInherited:
    """
    Применяет dir() для сбора атрибутов экземпляра и имен, унаследованных
    из его классов;
    в Python 3.X отображается больше имен, чем в Python 2.X из-за наличия
    подразумеваемого суперкласса object в модели классов нового стиля;
    getattr() извлекает унаследованные имена не в self.__dict__;
    используйте __str__, а не __repr__, иначе произойдет зацикливание при
    вызове связанных методов!
    """
    def __attrnames(self):
        result = ''
        for attr in dir(self):                          # dir() экземпляра
          if attr[:2] == '__' and attr[-2:] == '__': # Пропуск внутренних имен
                result += '\t%s\n' % attr
            else:
                result += '\t%s=%s\n' % (attr, getattr(self, attr))
        return result

    def __str__(self):
        return '<Instance of %s, address %s:\n%s>' % (
                        self.__class__.__name__,        # Имя класса
                        id(self),                       # Адрес
                        self.__attrnames())             # Список имя=значение

if __name__ == '__main__':
    import testmixin
    testmixin.tester(ListInherited)
```

Обратите внимание, что в коде пропускаются значения имен __X__; большинство из них являются внутренними именами, о которых мы обычно не заботимся в обобщенных списках подобного рода. В данной версии также должна использоваться встроенная функция `getattr` для извлечения атрибутов по строковым именам вместо индексации словаря атрибутов экземпляра — `getattr` задействует протокол поиска в иерархии наследования, а ряд имен, помещаемых здесь в список, не хранятся в самом экземпляре.

Чтобы протестировать новую версию, запустите ее файл напрямую — он передает тестовой функции из файла `testmixin.py` определяемый в нем класс для применения в качестве подмешиваемого в подклассе. Однако выходные данные тестовой функции и класса, выводящего список атрибутов, варьируются от выпуска к выпуску, т.к. результаты `dir` отличаются. В Python 2.X мы получаем следующий вывод; в имени метода класса для вывода списка атрибутов легко заметить корректировку имен в действии (мне пришлось усечь некоторые полные значения, чтобы они уместились на печатной странице):

```
c:\code> c:\python27\python listinherited.py
<Instance of Sub, address 35161352:
      _ListInherited__attrnames=<bound method Sub.__attrnames of <test...
не показано...>>
      __doc__
      __init__
      __module__
      __str__
      data1=spam
      data2=eggs
      data3=42
      ham=<bound method Sub.ham of <testmixin.Sub instance at 0x00000...
не показано...>>
      spam=<bound method Sub.spam of <testmixin.Sub instance at 0x00000...
не показано...>>
>
```

В Python 3.X отображается больше атрибутов, потому что все классы относятся к "новому стилю" и наследуют имена из подразумеваемого суперкласса `object`; более подробно об этом пойдет речь в главе 32. Поскольку из стандартного суперкласса наследуется настолько много имен, часть имен здесь не показаны — в Python 3.7 их в сумме 32. Запустите файл самостоятельно, чтобы получить полный список:

```
c:\code> c:\python37\python listinherited.py
<Instance of Sub, address 48032432:
  _ListInherited__attrnames=<bound method ListInherited.__attrnames of
<testmixin...не показано...>>
   __class__
   __delattr__
   __dict__
   __dir__
   __doc__
   __eq__
   ...остальные из 32 имен не показаны...
   __repr__
   __setattr__
   __sizeof__
   __str__
   __subclasshook__
```

```
  __weakref__
  data1=spam
  data2=eggs
  data3=42
  ham=<bound method tester.<locals>.Super.ham of <testmixin.
tester.<locals>.Sub object at 0x02DCEAB0>>
  spam=<bound method tester.<locals>.Sub.spam of <testmixin.
tester.<locals>.Sub object at 0x02DCEAB0>>
>
```

Как одно возможное усовершенствование, направленное на решение проблемы с ростом количества унаследованных имен и длинных значений, в следующей альтернативной версии `__attrnames` из файла `listinherited2.py` в пакете примеров для книги имена с двумя символами подчеркивания группируются отдельно, а переносы строк для длинных значений атрибутов сводятся к минимуму. Обратите внимание на отмену `%` с помощью `%%`, так что остается только один символ для финальной операции форматирования:

```
def __attrnames(self, indent=' '*4):
    result = 'Unders%s\n%s%%s\nOthers%s\n' % ('-'*77, indent, '-'*77)
    unders = []
    for attr in dir(self):                              # dir() экземпляра
        if attr[:2] == '__' and attr[-2:] == '__':    # Пропуск внутренних имен
            unders.append(attr)
        else:
            display = str(getattr(self, attr))[:82-(len(indent) + len(attr))]
            result += '%s%s=%s\n' % (indent, attr, display)
    return result % ', '.join(unders)
```

Благодаря такому изменению тестовый вывод класса становится чуть сложнее, но также более компактным и полезным:

```
c:\code> c:\python27\python listinherited2.py
<Instance of Sub, address 36299208:
Unders-----------------------------------------------------------------------------
    __doc__, __init__, __module__, __str__
Others-----------------------------------------------------------------------------
    _ListInherited__attrnames=<bound method Sub.__attrnames of <testmixin.
Sub insta
    data1=spam
    data2=eggs
    data3=42
    ham=<bound method Sub.ham of <testmixin.Sub instance at
0x000000000229E1C8>>
    spam=<bound method Sub.spam of <testmixin.Sub instance at
0x000000000229E1C8>>
>

c:\code> c:\python37\python listinherited2.py
<Instance of Sub, address 48159472:
Unders-----------------------------------------------------------------------------
    __class__, __delattr__, __dict__, __dir__, __doc__, __eq__, __format__, __ge__,
__getattribute__, __gt__, __hash__, __init__, __init_subclass__, __le__, __lt__,
__module__, __ne__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__,
__sizeof__, __str__, __subclasshook__, __weakref__
Others-----------------------------------------------------------------------------
```

```
   _ListInherited__attrnames=<bound method ListInherited.__attrnames of
<testmixin
   data1=spam
   data2=eggs
   data3=42
   ham=
<bound method tester.<locals>.Super.ham of <testmixin.tester.<locals>.Sub o
   spam=
<bound method tester.<locals>.Sub.spam of <testmixin.tester.<locals>.Sub o
>
```

Формат отображения – открытая для решения задача (например, стандартный модуль Python для "симпатичного вывода" pprint тоже способен предложить варианты), а потому дальнейшее совершенствование оставлено в качестве упражнения. Так или иначе, но класс, выводящий атрибуты в дереве классов, может оказаться более полезным.

Зацикливание в `__repr__`. Одно предостережение – теперь, когда мы отображаем также и унаследованные методы, для перегрузки операции вывода вместо `__repr__` должен использоваться метод `__str__`. В случае применения `__repr__` код попадет в бесконечный *цикл рекурсии* – отображение значения метода запускает `__repr__` класса этого метода, чтобы отобразить сам класс. То есть, если `__repr__` класса, выводящего список атрибутов, попытается отобразить метод, то класс отображаемого метода снова запустит `__repr__` класса, выводящего список атрибутов. Проблема тонкая, но реальная! Чтобы удостовериться в ее наличии, измените `__str__` на `__repr__`. Если вы обязаны использовать `__repr__` в контексте подобного рода, тогда избежать циклов можно за счет применения isinstance для сравнения типа значений атрибутов с types.MethodType из стандартной библиотеки и пропуска таких атрибутов.

Вывод списка атрибутов для объектов в деревьях классов

Давайте займемся последним расширением. В текущем виде класс, выводящий список атрибутов, включает в него унаследованные имена, но не указывает, из каких классов они были получены. Тем не менее, как было показано в примере classtree.py ближе к концу главы 29, реализовать подъем по деревьям наследования классов в коде довольно легко. Приведенный далее подмешиваемый класс (файл listtree.py) задействует ту же самую методику для отображения атрибутов, сгруппированных по классам, где они находятся – он схематически выводит полное *физическое дерево классов*, в ходе дела отображая атрибуты, которые присоединены к каждому объекту. Читателю все еще придется делать предположения относительно наследования атрибутов, но результат предоставляет гораздо больше деталей, чем простой плоский список:

```
#!python
# Файл listtree.py (Python 2.X + 3.X)

class ListTree:
    """
    Подмешиваемый класс, который возвращает в __str__ результат обхода целого
    дерева классов и атрибуты всех его объектов, начиная с self и выше;
    запускается print() и str() и возвращает сформированную строку;
    использует схему именования атрибутов __X, чтобы избежать конфликтов
    имен в клиентах; явно рекурсивно обращается к суперклассам,
    для ясности применяет str.format().
    """
```

```
    def __attrnames(self, obj, indent):
        spaces = ' ' * (indent + 1)
        result = ''
        for attr in sorted(obj.__dict__):
            if attr.startswith('__') and attr.endswith('__'):
                result += spaces + '{0}\n'.format(attr)
            else:
                result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
        return result

    def __listclass(self, aClass, indent):
        dots = '.' * indent
        if aClass in self.__visited:
            return '\n{0}<Class {1}:, address {2}: (see above)>\n'.format(
                            dots,
                            aClass.__name__,
                            id(aClass))
        else:
            self.__visited[aClass] = True
            here = self.__attrnames(aClass, indent)
            above = ''
            for super in aClass.__bases__:
                above += self.__listclass(super, indent+4)
            return '\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format(
                            dots,
                            aClass.__name__,
                            id(aClass),
                            here, above,
                            dots)

    def __str__(self):
        self.__visited = {}
        here = self.__attrnames(self, 0)
        above = self.__listclass(self.__class__, 4)
        return '<Instance of {0}, address {1}:\n{2}{3}>'.format(
                            self.__class__.__name__,
                            id(self),
                            here, above)

if __name__ == '__main__':
    import testmixin
    testmixin.tester(ListTree)
```

Класс `ListTree` достигает своей цели путем обхода дерева наследования – он начинает с `__class__` экземпляра, затем рекурсивно проходит по всем его суперклассам, перечисленным в `__bases__` класса, и попутно просматривает атрибут `__dict__` каждого объекта. При раскрутке рекурсии он добавляет каждую порцию дерева к результирующей строке.

Понимание рекурсивных программ подобного рода может требовать определенного времени, но с учетом произвольной формы и глубины деревьев классов в действительности у нас нет выбора (кроме реализации явных эквивалентов со стеком вроде представленных в главах 19 и 25 первого тома, которые оказываются не намного проще и ради экономии места здесь не приводятся). Однако для максимальной понятности код класса `ListTree` написан так, чтобы сделать его работу как можно более ясной.

Скажем, вы могли бы заменить оператор цикла из метода `__listclass`, показанный ниже в первом фрагменте кода, неявно запускаемым генераторным выражением, которое показано во втором фрагменте. Но второй фрагмент кода выглядит излишне запутанным в этом контексте (*рекурсивные вызовы*, внедренные в *генераторное выражение*) и не обеспечивает явного преимущества в плане производительности, особенно принимая во внимание ограниченные рамки данной программы (ни одна из альтернатив не создает временный список, хотя первая могла бы создавать больше временных результатов в зависимости от внутренней реализации строк, конкатенации и `join` — то, что потребует измерения времени посредством инструментов из главы 21 первого тома):

```
            above = ''
            for super in aClass.__bases__:
              above += self.__listclass(super, indent+4)
...или...
            above = ''.join(
              self.__listclass(super, indent+4) for super in aClass.__bases__)
```

Вы могли бы также реализовать конструкцию `else` в `__listclass` так, как показано ниже и делалось в предыдущем издании книги. Такая альтернативная версия помещает все в список аргументов `format`, полагается на тот факт, что вызов `join` запускает генераторное выражение и его рекурсивные вызовы *до* того, как операция форматирования начнет формировать результирующий текст, и выглядит более сложной для понимания:

```
self.__visited[aClass] = True
genabove = (self.__listclass(c, indent+4) for c in aClass.__bases__)
return '\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format(
                dots,
                aClass.__name__,
                id(aClass),
                self.__attrnames(aClass, indent),    # Запускается перед
                                                     #   форматированием!
                ''.join(genabove),
                dots)
```

Как всегда, явная реализация лучше неявной, и сам ваш код может быть не менее важным фактором, чем используемые в нем инструменты.

Также обратите внимание на применение в этой версии строкового метода `format` из Python 3.X и Python 2.6/2.7 вместо выражений форматирования `%` в попытке сделать подстановки яснее; при настолько большом количестве подстановок явные номера аргументов способны облегчить восприятие кода. Короче говоря, в данной версии мы заменили первую строку второй:

```
        return '<Instance of %s, address %s:\n%s%s>' % (...)       # Выражение
        return '<Instance of {0}, address {1}:\n{2}{3}>'.format(...)  # Метод
```

Запуск класса, выводящего дерево

Для тестирования необходимо запустить файл модуля с этим классом, как делалось ранее; он передает сценарию `testmixin.py` класс `ListTree`, чтобы подмешать его в подкласс в тестовой функции. Вот вывод, полученный в Python 2.X:

```
c:\code> c:\python27\python listtree.py
<Instance of Sub, address 36690632:
 _ListTree__visited={}
```

```
 data1=spam
 data2=eggs
 data3=42

....<Class Sub, address 36652616:
    __doc__
    __init__
    __module__
    spam=<unbound method Sub.spam>

........<Class Super, address 36652712:
        __doc__
        __init__
        __module__
        ham=<unbound method Super.ham>
........>

........<Class ListTree, address 30795816:
        _ListTree__attrnames=<unbound method ListTree.__attrnames>
        _ListTree__listclass=<unbound method ListTree.__listclass>
        __doc__
        __module__
        __str__
........>
....>
>
```

Обратите внимание в выводе на то, что теперь в Python 2.X методы являются *несвязанными*, т.к. мы напрямую извлекаем их из *классов*. В версии из предыдущего раздела они отображались как *связанные* методы, потому что ListInherited взамен извлекал их из *экземпляров* с помощью getattr (первая версия индексировала словарь __dict__ экземпляра и вообще не отображала методы, унаследованные из классов). Кроме того, таблица __visited из класса, выводящего дерево, в словаре атрибутов экземпляра имеет скорректированное имя; если только мы не крайне невезучи, то оно не будет конфликтовать с другими данными. Имена некоторых методов класса, выводящего дерево, также скорректированы, чтобы сделать их псевдозакрытыми.

Как показано ниже, в Python 3.X мы снова получаем добавочные атрибуты, которые могут варьироваться внутри линейки Python 3.X, и дополнительные суперклассы – в следующей главе вы узнаете, что в Python 3.X все классы верхнего уровня автоматически наследуются от встроенного класса object; в классах Python 2.X можно делать то же самое вручную, если для них выбрано поведение классов нового стиля. Также обратите внимание, что атрибуты, которые в Python 2.X были несвязанными методами, в Python 3.X представляют собой простые *функции*, как объяснялось ранее в главе (здесь снова ради экономии пространства удалены многие встроенные атрибуты класса object; запустите listtree.py самостоятельно, чтобы получить их полный список):

```
c:\code> c:\python37\python listtree.py
<Instance of Sub, address 48294960:
 _ListTree__visited={}
 data1=spam
 data2=eggs
 data3=42

....<Class Sub, address 48361520:
    __doc__
    __init__
```

```
    __module__
    spam=<function tester.<locals>.Sub.spam at 0x02E1BC48>
........<Class Super, address 48319768:
        __dict__
        __doc__
        __init__
        __module__
        __weakref__
        ham=<function tester.<locals>.Super.ham at 0x02E1BBB8>
............<Class object, address 1465979880:
            __class__
            __delattr__
            __dir__
            __doc__
            __eq__
            ...остальные атрибуты не показаны: всего их 23...
            __repr__
            __setattr__
            __sizeof__
            __str__
            __subclasshook__
............>
........>
........<Class ListTree, address 14115808:
        _ListTree__attrnames=<function ListTree.__attrnames at 0x02E1B9C0>
        _ListTree__listclass=<function ListTree.__listclass at 0x02E1BA08>
        __dict__
        __doc__
        __module__
        __str__
        __weakref__
............<Class object:, address 1465979880: (see above)>
........>
....>
>
```

В этой версии устранена возможность двукратного вывода одного и того же объекта класса за счет ведения таблицы *посещенных* классов (вот почему включается id объекта — чтобы служить ключом для ранее отображенного элемента в дереве). Подобно инструменту транзитивной перегрузки модулей из главы 25 первого тома словарь помогает избежать повторений в выводе, потому что объекты классов являются хешируемыми и могут быть ключами словаря; множество обеспечило бы аналогичную функциональность.

Формально циклы в деревьях наследования классов обычно невозможны. Класс должен быть уже определен, чтобы его можно было указывать в качестве суперкласса, и Python сгенерирует исключение, если вы попытаетесь позже создать цикл за счет изменения `__bases__`, но механизм учета посещенных классов не допускает повторного вывода класса:

```
>>> class C: pass
>>> class B(C): pass
>>> C.__bases__ = (B,)       # Черная магия!
TypeError: a __bases__ item causes an inheritance cycle
Ошибка типа: элемент __bases__ вызывает цикл наследования
```

Вариант использования: отображение значений имен с символами подчеркивания

Текущая версия также избегает отображения крупных внутренних объектов, снова пропуская имена __X__. Если вы поместите в комментарии код, который трактует такие имена особым образом:

```
        for attr in sorted(obj.__dict__):
#           if attr.startswith('__') and attr.endswith('__'):
#              result += spaces + '{0}\n'.format(attr)
#           else:
              result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
```

то их значения станут нормально отображаться. Ниже приведен вывод в Python 2.X после внесения этого временного изменения, содержащий значения всех атрибутов в дереве классов:

```
c:\code> c:\python27\python listtree.py
<Instance of Sub, address 35750408:
 _ListTree__visited={}
 data1=spam
 data2=eggs
 data3=42

....<Class Sub, address 36353608:
    __doc__=None
    __init__=<unbound method Sub.__init__>
    __module__=testmixin
    spam=<unbound method Sub.spam>

........<Class Super, address 36353704:
        __doc__=None
        __init__=<unbound method Super.__init__>
        __module__=testmixin
        ham=<unbound method Super.ham>
........>

........<Class ListTree, address 31254568:
        _ListTree__attrnames=<unbound method ListTree.__attrnames>
        _ListTree__listclass=<unbound method ListTree.__listclass>
        __doc__=
Подмешиваемый класс, который возвращает в __str__ результат обхода целого
дерева классов и атрибуты всех его объектов, начиная с self и выше;
запускается print() и str() и возвращает сформированную строку; использует
схему именования атрибутов __X, чтобы избежать конфликтов имен в клиентах;
явно рекурсивно обращается к суперклассам, для ясности применяет str.format().

        __module__=__main__
        __str__=<unbound method ListTree.__str__>
........>
....>
>
```

Вывод теста в Python 3.X оказывается гораздо более длинным и в целом может оправдать отделение имен с символами подчеркивания, которое делалось ранее:

```
c:\code> c:\python37\python listtree.py
<Instance of Sub, address 47901712:
 _ListTree__visited={}
 data1=spam
```

```
 data2=eggs
 data3=42
....<Class Sub, address 47968304:
    __doc__=None
    __init__=<function tester.<locals>.Sub.__init__ at 0x02DBBC00>
    __module__=testmixin
    spam=<function tester.<locals>.Sub.spam at 0x02DBBC48>
........<Class Super, address 47922456:
        __dict__={'__module__': 'testmixin', '__init__': <function
tester.<locals>.Super.__init__ at 0x02DBBB70>, 'ham': <function
tester.<locals>.Super.ham at 0x02DBBBB8>, '__dict__': <attribute '__dict__'
of 'Super' objects>, '__weakref__': <attribute '__weakref__' of 'Super'
objects>, '__doc__': None}
        __doc__=None
        __init__=<function tester.<locals>.Super.__init__ at 0x02DBBB70>
        __module__=testmixin
        __weakref__=<attribute '__weakref__' of 'Super' objects>
        ham=<function tester.<locals>.Super.ham at 0x02DBBBB8>
............<Class object, address 1465979880:
            __class__=<class 'type'>
            __delattr__=<slot wrapper '__delattr__' of 'object' objects>
            __dir__=<method '__dir__' of 'object' objects>
            __doc__=The most base type
            __eq__=<slot wrapper '__eq__' of 'object' objects>
            __format__=<method '__format__' of 'object' objects>
            __ge__=<slot wrapper '__ge__' of 'object' objects>
            __getattribute__=
              <slot wrapper '__getattribute__' of 'object' objects>
            __gt__=<slot wrapper '__gt__' of 'object' objects>
            __hash__=<slot wrapper '__hash__' of 'object' objects>
            __init__=<slot wrapper '__init__' of 'object' objects>
            __init_subclass__=
             <built-in method __init_subclass__ of type object at 0x576113E8>
            __le__=<slot wrapper '__le__' of 'object' objects>
            __lt__=<slot wrapper '__lt__' of 'object' objects>
            __ne__=<slot wrapper '__ne__' of 'object' objects>
            __new__=<built-in method __new__ of type object at 0x576113E8>
            __reduce__=<method '__reduce__' of 'object' objects>
            __reduce_ex__=<method '__reduce_ex__' of 'object' objects>
            __repr__=<slot wrapper '__repr__' of 'object' objects>
            __setattr__=<slot wrapper '__setattr__' of 'object' objects>
            __sizeof__=<method '__sizeof__' of 'object' objects>
            __str__=<slot wrapper '__str__' of 'object' objects>
            __subclasshook__=
              <built-in method __subclasshook__ of type object at 0x576113E8>
............>
........>
........<Class ListTree, address 46293984:
        _ListTree__attrnames=<function ListTree.__attrnames at 0x02DBB9C0>
        _ListTree__listclass=<function ListTree.__listclass at 0x02DBBA08>
        __dict__={'__module__': '__main__', '__doc__': "\n    Подмешиваемый
класс, который возвращает в __str__ результат обхода целого дерева классов
\n и атрибуты всех его объектов, начиная с selfи выше; запускается print()
и str() \n и возвращает сформированную строку; использует схему именования
```

```
атрибутов __X, \n чтобы избежать конфликтов имен в клиентах; явно рекурсивно
обращается к суперклассам, \n для ясности применяет str.format().\n    ",
'_ListTree__attrnames': <function ListTree.__attrnames at 0x02DBB9C0>,
'_ListTree__listclass': <function ListTree.__listclass at 0x02DBBA08>,
'__str__': <function ListTree.__str__ at 0x02DBBA50>,
'__dict__': <attribute '__dict__'
of 'ListTree' objects>, '__weakref__': <attribute '__weakref__'
of 'ListTree' objects>}
        __doc__=
    Подмешиваемый класс, который возвращает в __str__ результат обхода
целого дерева классов и атрибуты всех его объектов, начиная с self и выше;
запускается print() и str() и возвращает сформированную строку; использует
схему именования атрибутов __X, чтобы избежать конфликтов имен в клиентах;
явно рекурсивно обращается к суперклассам, для ясности применяет str.format().

        __module__=__main__
        __str__=<function ListTree.__str__ at 0x02DBBA50>
        __weakref__=<attribute '__weakref__' of 'ListTree' objects>
............<Class object:, address 1465979880: (see above)>
........>
....>
>
```

Вариант использования: запуск для более крупных модулей

Ради интереса удалите комментарии в строках обработки имен с символами подчеркивания и попробуйте подмешать данный класс во что-то более массивное, скажем, в класс `Button` из модуля Python с комплектом инструментов для построения графических пользовательских интерфейсов `tkinter`. В общем случае вам понадобится указать имя класса `ListTree` первым (крайним слева) в заголовке `class`, чтобы выбирался его метод `__str__`; класс `Button` тоже имеет такой метод, но при множественном наследовании первым всегда выполняется поиск в крайнем слева суперклассе.

Вывод получается довольно большим (более 18 тысяч символов и 320 строк в Python 3.X – и почти 35 тысяч символов и 336 строк, если вы забыли удалить комментарий с кода обнаружения символов подчеркивания!), поэтому запустите код, чтобы увидеть полный список. Обратите внимание, что атрибут словаря `__visited` нашего класса, выводящего дерево, безвредно смешивается с атрибутами, которые создал сам модуль `tkinter`. Если вы работаете с Python 2.X, то также не забудьте применять имя модуля `Tkinter` вместо `tkinter`:

```
>>> from listtree import ListTree
>>> from tkinter import Button                    # Оба класса имеют метод __str__
>>> class MyButton(ListTree, Button): pass        # ListTree первый: используется
                                                  #   его метод __str__
>>> B = MyButton(text='spam')
>>> open('savetree.txt', 'w').write(str(B))  # Сохранить файл для просмотра
                                                  #   в будущем
18497
>>> len(open('savetree.txt').readlines())         # Строк в файле
320
>>> print(B)                                      # Вывод всего дерева
<Instance of MyButton, address 47980912:
 _ListTree__visited={}
 _name=!mybutton
```

```
 _tclCommands=[]
 _w=.!mybutton
 children={}
 master=.
...очень многое не показано...
>
>>> S = str(B)                          # Или вывод только первой части
>>> print(S[:1000])
<Instance of MyButton, address 47980912:
 _ListTree__visited={}
 _name=!mybutton
 _tclCommands=[]
 _w=.!mybutton
 children={}
 master=.
 tk=<_tkinter.tkapp object at 0x02DFAF08>
 widgetName=button

....<Class MyButton, address 48188696:
    __doc__
    __module__

........<Class ListTree, address 46687200:
        _ListTree__attrnames=<function ListTree.__attrnames at 0x02DFBA50>
        _ListTree__listclass=<function ListTree.__listclass at 0x02DFBA98>
        __dict__
        __doc__
        __module__
        __str__
        __weakref__

............<Class object, address 1465979880:
            __class__
            __delattr__
            __dir__
            __doc__
            __eq__
            __format__
            __ge__
            __getattribute__
            __gt__
            __hash__
            __init__
            __init_subclass__
            __le__
            __lt__
            __ne__
            __new__
            __reduce__
```

Поэкспериментируйте с кодом самостоятельно. Суть здесь в том, что ООП направлено на многократное использование кода, а подмешиваемые классы являются ярким примером. Как и почти все остальное в программировании, множественное наследование при надлежащем применении может быть полезным механизмом. Тем не менее, на практике оно представляет собой развитое средство и может стать сложным в случае небрежного или чрезмерного использования. Мы возвратимся к данной теме, когда будем рассматривать затруднения в конце следующей главы.

Собирающий модуль

Наконец, чтобы еще больше облегчить импортирование наших инструментов, мы можем предоставить собирающий модуль, который объединяет их в единое пространство имен — импортирование только одного этого модуля открывает доступ сразу ко всем трем подмешиваемым классам:

```
# Файл lister.py
# Для удобства собирает в одном модуле все три класса, выводящие атрибуты

from listinstance import ListInstance
from listinherited import ListInherited
from listtree import ListTree

Lister = ListTree    # Выбрать стандартный класс, выводящий атрибуты
```

Импортеры могут работать с индивидуальными именами классов или назначать им псевдоним в виде общего имени, применяемом в подклассах, которое можно модифицировать в операторе import:

```
>>> import lister
>>> lister.ListInstance   # Использовать специфический класс, выводящий атрибуты
<class 'listinstance.ListInstance'>
>>> lister.Lister                # Использовать стандартный класс Lister
<class 'listtree.ListTree'>

>>> from lister import Lister      # Использовать стандартный класс Lister
>>> Lister
<class 'listtree.ListTree'>

>>> from lister import ListInstance as Lister  # Использовать псевдоним Lister
>>> Lister
<class 'listinstance.ListInstance'>
```

Python часто делает гибкие API-интерфейсы инструментов почти автоматическими.

Возможности для совершенствования: MRO, слоты, графические пользовательские интерфейсы

Подобно большинству программ есть очень многое, что мы могли бы еще сделать здесь. Далее приведены советы по расширению, которые вы можете счесть полезными. Некоторые из них выливаются в интересные проекты, а два служат плавным переходом к материалу следующей главы, но из-за ограниченного пространства они оставлены в качестве упражнений для самостоятельного выполнения.

Общие идеи: графические пользовательские интерфейсы, внутренние имена

Группирование имен с двумя символами подчеркивания, как делалось ранее, может способствовать сокращению размера древовидного отображения, хотя некоторые имена вроде `__init__` определяются пользователем и заслуживают специального обращения. Схематическое изображение дерева в графическом пользовательском интерфейсе тоже может считаться естественным следующим шагом — комплект инструментов `tkinter`, задействованный в примерах из предыдущего раздела, поставляется вместе с Python и предлагает базовую, но легкую поддержку, а есть альтернативы с более широкими возможностями, хотя они сложнее. Дополнительные указания по этому поводу ищите в разделе “Указания на будущее” главы 28.

Физические деревья или наследование: использование MRO (предварительный обзор)

В следующей главе мы также обсудим модель классов нового стиля, которая модифицирует порядок поиска для одного особого случая множественного наследования (ромбы). Там мы также исследуем атрибут объектов классов нового стиля `class.__mro__` – кортеж, который дает применяемый при наследовании порядок поиска в дереве классов, известный как MRO нового стиля.

В текущем виде наш класс `ListTree` схематически отображает *физическую форму* дерева наследования и ожидает, что пользователь сделает вывод о том, откуда унаследован тот или иной атрибут. В этом заключалась его цель, но универсальное средство просмотра объектов могло бы также использовать кортеж MRO, чтобы автоматически ассоциировать атрибут с классом, из которого он *унаследован.* За счет обследования MRO нового стиля (или упорядочение DFLR классических классов) для каждого унаследованного атрибута в результате вызова `dir` мы можем эмулировать поиск в иерархии наследования Python и сопоставлять атрибуты с их исходными объектами в отображенном физическом дереве классов.

На самом деле мы *будем* писать код, который очень близок к этой идее, в модуле `mapattrs` из следующей главы и многократно применять его тестовые классы для демонстрации идеи, так что дождитесь эпилога данной истории. Он мог бы использоваться взамен или в дополнение к отображению физических местоположений атрибутов в `__attrnames`; обе формы снабжали бы программистов полезными сведениями. Такой подход позволяет также учитывать слоты, которые рассматриваются в следующем примечании.

Виртуальные данные: слоты, свойства и многое другое (предварительный обзор)

Поскольку классы `ListInstance` и `ListTree` просматривают словари пространств имен `__dict__` экземпляров, они представлены здесь для иллюстрации ряда тонких проблем проектирования. В классах Python некоторые имена, ассоциированные с данными экземпляра, могут не храниться в самом экземпляре. Сюда входят свойства нового стиля, слоты и дескрипторы, представленные в следующей главе, а также атрибуты, динамически вычисляемые во всех классах с помощью инструментов вроде `__getattr__`. Имена упомянутых “виртуальных” атрибутов не хранятся в словаре пространства имен экземпляра, поэтому ни одно из них не будет отображаться как часть собственных данных экземпляра.

Из всего перечисленного *слоты* кажутся наиболее тесно связанными с экземпляром; они хранят данные в экземплярах, хотя их имена не появляются в словарях пространств имен экземпляров. Свойства и дескрипторы тоже ассоциированы с экземплярами, но не занимают место в экземпляре, их вычислительная природа гораздо более явная и они могут показаться ближе к методам уровня класса, чем данные экземпляров.

Как мы увидим в следующей главе, слоты функционируют подобно атрибутам экземпляров, но создаются и управляются автоматически создаваемыми элементами в классах. Они являются относительно редко применяемым вариантом классов нового стиля, где атрибуты экземпляров объявляются в атрибуте класса `__slots__` и физически не хранятся в словаре `__dict__` экземпляра; на самом деле слоты могут вообще подавлять `__dict__`. По указанной причине инструменты, которые отображают экземпляры путем просмотра только их пространств имен, не будут напрямую связывать экземпляр с атрибутами, хра-

нящимися в слотах. В существующем виде класс ListTree отображает слоты как атрибуты класса, когда бы они ни появлялись (хотя не относит их к экземпляру), а класс ListInstance вообще их не отображает.

Хотя это обретет больше смысла после изучения самого средства в следующей главе, оно оказывает воздействие на код здесь и на похожие инструменты. Скажем, если в textmixin.py мы присвоим __slots__=['data1'] в Super и __slots__=['data3'] в Sub, то два класса, выводящие дерево, отобразят в экземпляре только атрибут data2. Класс ListTree также отобразит data1 и data3, но как атрибуты объектов *классов* Super и Sub и со специальным форматом для их значений (формально они являются дескрипторами уровня класса — еще одним инструментом нового стиля, представляемым в главе 32).

В следующей главе объясняется, что для отображения атрибутов из слотов как имен экземпляров инструменты обычно должны использовать dir, чтобы получить список всех атрибутов — физически присутствующих и унаследованных — и затем применять либо getattr для извлечения их значений из экземпляра, либо __dict__ в обходах дерева для извлечения значений из их источника наследования и принимать отображение реализаций некоторых из них в классах. Поскольку dir включает имена унаследованных "виртуальных" атрибутов (в том числе слоты и свойства), они попадут в набор экземпляра. Как также обнаружится, MRO может содействовать здесь для сопоставления атрибутов dir с их источниками или для ограничения отображения экземпляров именами, записанными в определяемых пользователем классах, за счет отсеивания имен, которые унаследованы из встроенного класса object.

Класс ListInherited невосприимчив к большинству этого, т.к. он уже отображает полный результирующий набор dir, который содержит имена __dict__ и имена __slots__ всех классов, хотя в существующем виде его использование минимально. Вариант ListTree, употребляющий методику с dir наряду с последовательностью MRO для сопоставления атрибутов с классами, будет применяться также к слотам, потому что основанные на слотах имена появляются в результатах __dict__ класса по отдельности как инструменты управления слотами, но не в __dict__ экземпляра.

Альтернативно в качестве политики мы могли бы просто позволить коду обрабатывать основанные на слотах атрибуты так, как он делает в текущий момент, вместо того, чтобы усложнять его для учета редко используемого и продвинутого средства, которое в наши дни даже считается сомнительной практикой. Слоты и нормальные атрибуты экземпляров являются разными видами имен. В действительности отображение имен слотов как атрибутов классов, а не экземпляров, формально будет более точным — в следующей главе мы увидим, что хотя их реализация находится в классах, занимаемое ими пространство располагается в экземплярах.

В конечном итоге попытка собрать все "виртуальные" атрибуты, ассоциированные с классом, может оказаться чем-то вроде несбыточной мечты. Обрисованные здесь методики способны решить задачу со слотами и свойствами, но некоторые атрибуты являются *полностью* динамическими, вообще не имея какой-либо физической основы: атрибуты, вычисляемые при извлечении обобщенным методом наподобие __getattr__, не относятся к данным в классическом смысле. Инструменты, которые пытаются отображать данные в на-

столько динамическом языке, как Python, должны сопровождаться предупреждением о том, что отдельные данные *в лучшем случае нематериальны*[3]!

Мы также предложим небольшое расширение кода, представленного в настоящем разделе, в упражнениях в конце этой части книги, чтобы выводить имена суперклассов в круглых скобках в начале отображения экземпляров. Для лучшего понимания предшествующих двух пунктов нам необходимо завершить текущую главу и перейти к следующей и последней в данной части книги.

Другие темы, связанные с проектированием

В этой главе мы исследовали наследование, композицию, делегирование, множественное наследование, связанные методы и фабрики – все распространенные паттерны проектирования, применяемые для объединения классов в программах на Python. Но, по правде говоря, мы лишь слегка коснулись поверхности предметной области, связанной с паттернами проектирования. В других местах книги вы обнаружите обзор остальных тем, относящихся к проектированию, таких как:

- абстрактные суперклассы (глава 29);
- декораторы (главы 32 и 39);
- подклассы типов (глава 32);
- статические методы и методы классов (глава 32);
- управляемые атрибуты (главы 32 и 38);
- метаклассы (главы 32 и 40).

Тем не менее, за дополнительными сведениями о паттернах проектирования я рекомендую обратиться к другим источникам, посвященным ООП в целом. Хотя паттерны важны и зачастую более естественны в ООП на Python, чем на других языках, они не являются специфическими для самого языка Python, а представляют собой предмет, который лучше всего усваивается с опытом.

Резюме

В главе рассматривались избранные способы использования и комбинирования классов с целью оптимизации возможности их многократного применения и приобретения других преимуществ – то, что обычно считается задачами проектирования, которые часто независимы от какого-либо конкретного языка программирования (хотя Python способен облегчить их реализацию). Вы изучили *делегирование* (помещение объектов внутрь промежуточных классов), *композицию* (управление внедренными объектами) и *наследование* (получение линий поведения от других классов), а также ряд более экзотических концепций, таких как псевдозакрытые атрибуты, множественное наследование, связанные методы и фабрики.

---

[3] Некоторые динамические и промежуточные объекты, основанные на `__getattr__` и подобных приемах, могут также использовать метод перегрузки операции `__dir__`, чтобы вручную публиковать список атрибутов для вызовов `dir`. Однако поскольку это необязательно, универсальные инструменты не должны рассчитывать на то, что их клиентские классы поступят так. Дополнительную информацию о методе `__dir__` ищите в книге *Python Pocket Reference*, `http://www.oreilly.com/catalog/9780596009403/` (*Python. Карманный справочник*, 5-е изд., `http://www.williamspublishing.com/Books/978-5-907114-60-9.html`).

В следующей главе мы завершаем исследование классов и ООП обзором более сложных тем, связанных с классами. Определенные материалы могут быть интереснее разработчикам инструментов, нежели прикладным программистам, но они заслуживают ознакомления большинством из тех, кто занимается ООП на Python — если уж не для своего кода, то для кода, написанного другими, в котором необходимо разобраться. Но сначала закрепите пройденный материал главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Что такое множественное наследование?
2. Что такое делегирование?
3. Что такое композиция?
4. Что такое связанные методы?
5. Для чего используются псевдозакрытые атрибуты?

Проверьте свои знания: ответы

1. Множественное наследование происходит, когда класс наследуется от нескольких суперклассов; оно полезно для смешивания множества пакетов кода, основанного на классах. Общий порядок поиска атрибутов определяется порядком слева направо, в котором суперклассы указаны в строках заголовка оператора `class`.

2. Делегирование подразумевает помещение объекта внутрь промежуточного класса, который добавляет дополнительное поведение и передает остальные операции внутреннему объекту. Промежуточный класс предохраняет интерфейс внутреннего объекта.

3. Композиция представляет собой методику, посредством которой класс контроллера внедряет в себя и управляет несколькими объектами, а также обеспечивает все собственные интерфейсы; она является способом построения более крупных структур с помощью классов.

4. Связанные методы объединяют экземпляр и функцию метода; их можно вызывать, не передавая явно объект экземпляра, поскольку исходный экземпляр по-прежнему доступен.

5. Псевдозакрытые атрибуты (чьи имена начинаются с двух символом подчеркивания, но не заканчиваются ими: `__X`) используются для локализации имен во включающем классе. К ним относятся атрибуты класса вроде методов, определенных внутри класса, и атрибуты экземпляра `self`, которым присваиваются значения в методах класса. Такие имена расширяются для включения имени класса, что делает их в целом уникальными.

ГЛАВА 32

Расширенные возможности классов

В этой главе описание ООП на Python завершается представлением нескольких более сложных тем, связанных с классами: мы рассмотрим создание подклассов из встроенных типов, изменения и расширения классов "нового стиля", статические методы и методы классов, слоты и свойства, декораторы функций и классов, MRO и встроенную функцию `super`, а также многое другое.

Как будет показано, модель ООП на Python в своей основе относительно проста и некоторые из обсуждаемых в главе средств настолько продвинуты и необязательны, что вы нечасто будете сталкиваться с ними в своей карьере прикладного программирования на Python. Однако в интересах полноты – и ввиду того, что никогда нельзя предугадать, какое "продвинутое" средство неожиданно обнаружится в используемом вами коде – мы закончим обсуждение классов кратким обзором сложных инструментов такого рода для ООП.

Поскольку глава является последней в данной части, в ее конце приводится раздел, посвященный связанным с классами затруднениям, а также подборка подходящих упражнений. Я призываю вас проработать предложенные упражнения, чтобы закрепить понимание идей, раскрываемых в главе. Я также предлагаю в качестве дополнения к материалам книги выполнить либо изучить более крупные объектно-ориентированные проекты на Python. Подобно многому в вычислительной обработке преимущества ООП становятся более очевидными при практическом применении.

Замечания по содержанию. В настоящей главе собраны более сложные темы, касающиеся классов, но некоторые из них слишком обширные, чтобы их можно было полностью раскрыть в одной главе. Такие темы, как свойства, дескрипторы, декораторы и метаклассы, упоминаются здесь лишь кратко, а более подробно рассматриваются в *финальной части* книги после исключений. Обязательно просмотрите более полные примеры и расширенное описание ряда тем, которые подпадают под категорию исследуемых в главе.

Вы также заметите, что эта глава самая объемная в книге – я предполагаю наличие у читателей достаточной смелости, чтобы засучив рукава, приступить к ее изучению. Если вас пока не интересуют расширенные темы по ООП, тогда можете перейти сразу к упражнениям в конце главы и вернуться к ее чтению в будущем, когда вы столкнетесь с такими инструментами в коде, с которым придется работать.

Расширение встроенных типов

Помимо реализации новых видов объектов классы иногда используются для расширения функциональности встроенных типов Python с целью поддержки более экзотических структур данных. Например, чтобы добавить к спискам методы вставки и удаления из очереди, вы можете реализовать классы, которые внедряют списковый объект и экспортируют методы вставки и удаления, обрабатывающие список особым образом, с помощью методики делегирования, описанной в главе 31. Начиная с версии Python 2.2, вы также можете применять наследование для специализации встроенных типов. В следующих двух разделах обе методики показаны в действии.

Расширение типов путем внедрения

Помните ли вы функции для работы с множествами, которые были написаны в главах 16 и 18 первого тома? Далее вы увидите, как они выглядят после возрождения в виде класса Python. В приведенном ниже примере (файл `setwrapper.py`) реализован новый объектный тип множества за счет переноса ряда функций для работы с множествами в методы и добавления перегрузки базовых операций. По большей части этот класс является всего лишь оболочкой для списка Python с дополнительными операциями над множествами. Но будучи классом, он также поддерживает многочисленные экземпляры и настройку путем наследования в подклассах. В отличие от наших ранних функций использование классов дает возможность создавать набор самодостаточных объектов с предварительно установленными данными и поведением, а не передавать функциям списки вручную:

```
class Set:
   def __init__(self, value = []):          # Конструктор
      self.data = []                        # Управляет списком
      self.concat(value)

   def intersect(self, other):              # other - любая последовательность
      res = []                              # self — подчиненный объект
      for x in self.data:
         if x in other:                     # Выбрать общие элементы
            res.append(x)
      return Set(res)                       # Возвратить новый объект Set

   def union(self, other):                  # other - любая последовательность
      res = self.data[:]                    # Копировать список
      for x in other:                       # Добавить элементы в other
         if not x in res:
            res.append(x)
      return Set(res)

   def concat(self, value):                 # value: список, Set...
      for x in value:                       # Удаляет дубликаты
         if not x in self.data:
            self.data.append(x)

def __len__(self): return len(self.data)    # len(self), если self истинно
def __getitem__(self, key): return self.data[key]       # self[i], self[i:j]
def __and__(self, other): return self.intersect(other)  # self & other
def __or__(self, other): return self.union(other)       # self | other
def __repr__(self): return 'Set:' + repr(self.data)     # print(self),...
def __iter__(self): return iter(self.data)              # for x in self,...
```

Для использования класса Set мы создаем экземпляры, вызываем методы и запускаем определенные операции обычным образом:

```
from setwrapper import Set
x = Set([1, 3, 5, 7])
print(x.union(Set([1, 4, 7])))      # Выводится Set:[1, 3, 5, 7, 4]
print(x | Set([1, 4, 6]))           # Выводится Set:[1, 3, 5, 7, 4, 6]
```

Перегрузка операций, таких как индексирование и итерация, также часто позволяет экземплярам нашего класса Set выдавать себя за реальные списки. Поскольку в упражнении, предложенном в конце главы, вы будете взаимодействовать и расширять этот класс, дальнейшее обсуждение кода откладывается до приложения Г.

Расширение типов путем создания подклассов

Начиная с версии Python 2.2, для всех встроенных типов в языке можно создавать подклассы напрямую. Функции преобразования типов вроде list, str, dict и tuple стали именами встроенных типов – несмотря на прозрачность для вашего сценария, вызов преобразования типа (скажем, list('spam')) теперь на самом деле представляет собой обращение к конструктору объектов типа.

Такое изменение предоставляет возможность настройки или расширения поведения встроенных типов посредством определяемых пользователем операторов class: для их настройки просто создавайте подклассы с новыми именами типов. Экземпляры подклассов ваших типов обычно могут применяться везде, где допускается появление исходных встроенных типов. Например, предположим, что вы испытываете трудности с привыканием к тому факту, что смещения в списках Python начинаются с 0, а не 1. Не переживайте – вы всегда можете создать собственный подкласс, который настраивает эту основную линию поведения списков, как демонстрируется в файле typesub-class.py:

```
# Создание подкласса встроенного типа/класса списка
# Отображает 1..N на 0..N-1; обращается к встроенной версии.
class MyList(list):
    def __getitem__(self, offset):
        print('(indexing %s at %s)' % (self, offset))
        return list.__getitem__(self, offset - 1)
if __name__ == '__main__':
    print(list('abc'))
    x = MyList('abc')              # Метод __init__, унаследованный от списка
    print(x)                       # Метод __repr__, унаследованный от списка

    print(x[1])                    # MyList.__getitem__
    print(x[3])                    # Настраивает метод из суперкласса списка

    x.append('spam'); print(x)     # Атрибуты из суперкласса списка
    x.reverse(); print(x)
```

Подкласс MyList расширяет только метод индексирования __getitem__ встроенного списка, чтобы отображать индексы 1-N на требующиеся 0-N-1. Он всего лишь декрементирует переданный индекс и вызывает версию индексирования из суперкласса, но этого достаточно для выполнения трюка:

```
% python typesubclass.py
['a', 'b', 'c']
['a', 'b', 'c']
(indexing ['a', 'b', 'c'] at 1)
a
```

```
(indexing ['a', 'b', 'c'] at 3)
c
['a', 'b', 'c', 'spam']
['spam', 'c', 'b', 'a']
```

Полученный вывод также включает трассировочный текст, который класс отображает при индексировании. Конечно, остался *другой вопрос*, является ли вообще хорошей идеей подобное изменение индексирования – пользователи класса `MyList` могут быть крайне озадачены довольно радикальным отступлением от поведения последовательностей в Python! Тем не менее, возможность такого рода настройки встроенных типов может рассматриваться как ценное качество.

Например, эта кодовая схема порождает альтернативный способ реализации множества – как подкласса встроенного спискового типа, а не автономного класса, который управляет внедренным объектом списка, как было показано в предыдущем разделе. Как было указано в главе 5 первого тома, в наши дни Python поступает с мощным встроенным объектом множества наряду с синтаксисом литералов и включений для создания новых множеств. Однако его самостоятельная реализация по-прежнему остается великолепным способом изучить создание подклассов для типов в целом.

Следующий класс из файла `setsubclass.py` настраивает списки, добавляя методы и операции для обработки множеств. Поскольку остальные линии поведения наследуются из встроенного суперкласса `list`, класс `Set` становится более короткой и простой альтернативой предыдущему варианту – все, что здесь не определено, направляется прямо `list`:

```
from __future__ import print_function  # Совместимость с Python 2.X
class Set(list):
    def __init__(self, value = []):     # Конструктор
        list.__init__(self)             # Настраивает список
        self.concat(value)      # Копирует изменяемые стандартные значения

    def intersect(self, other):         # other - любая последовательность
        res = []                        # self – подчиненный объект
        for x in self:
            if x in other:              # Выбрать общие элементы
                res.append(x)
        return Set(res)                 # Возвратить новый объект Set

    def union(self, other):             # other - любая последовательность
        res = Set(self)                 # Копировать текущий и другой список
        res.concat(other)
        return res

    def concat(self, value):            # value: список, Set и т.д.
        for x in value:                 # Удаляет дубликаты
            if not x in self:
                self.append(x)
    def __and__(self, other): return self.intersect(other)
    def __or__(self, other): return self.union(other)
    def __repr__(self): return 'Set:' + list.__repr__(self)

if __name__ == '__main__':
    x = Set([1,3,5,7])
    y = Set([2,1,4,5,6])
    print(x, y, len(x))
    print(x.intersect(y), y.union(x))
    print(x & y, x | y)
    x.reverse(); print(x)
```

Ниже приведен вывод кода самотестирования, находящегося в конце файла. Из-за того, что создание подклассов для основных типов — кое в чем сложное средство с ограниченной целевой аудиторией, дальнейшие детали здесь опущены, но я предлагаю вам отследить получение результатов в коде, чтобы изучить его поведение (одинаковое в Python 3.X и 2.X):

```
% python setsubclass.py
Set:[1, 3, 5, 7] Set:[2, 1, 4, 5, 6] 4
Set:[1, 5] Set:[2, 1, 4, 5, 6, 3, 7]
Set:[1, 5] Set:[1, 3, 5, 7, 2, 4, 6]
Set:[7, 5, 3, 1]
```

В Python существуют более эффективные способы реализации множеств с помощью словарей. Они заменяют вложенные линейные просмотры в показанных ранее реализациях множеств более прямыми операциями индексирования в словарях (хеширование) и потому выполняются гораздо быстрее. Подробности ищите в книге *Programming Python* (`http://www.oreilly.com/catalog/9780596158101`). Если вас интересуют множества, тогда снова взгляните на объектный тип `set`, который был исследован в главе 5 первого тома; он предоставляет обширный набор операций для работы с множествами в виде встроенных инструментов. С реализациями множеств забавно экспериментировать, но в современном Python они больше не являются строго обязательными.

За еще одним примером создания подклассов для типов обращайтесь к реализации типа `bool` в Python 2.3 и последующих версиях. Как упоминалось ранее в книге, тип `bool` реализован как подкласс `int` с двумя экземплярами (`True` и `False`), которые ведут себя подобно целым числам `1` и `0`, но наследуют специальные методы строкового представления, отображающие их имена.

Модель классов "нового стиля"

В выпуске Python 2.2 была введена новая разновидность классов, известная как классы *нового стиля*; когда с ними сравнивают классы, которые следуют первоначальной и традиционной модели, их называют *классическими* классами. В Python 3.X два вида классов были объединены, но для пользователей и кода на Python 2.X они остаются раздельными.

- В Python 3.X все классы являются тем, что прежде называлось "новым стилем", унаследованы они явно от `object` или нет. Указывать суперкласс `object` не обязательно, и он подразумевается.
- В Python 2.X классы должны явно наследоваться от `object` (или другого встроенного типа), чтобы считаться "новым стилем" и получить все линии поведения нового стиля. Без такого наследования классы будут "классическими".

Поскольку в Python 3.X все классы автоматически становятся классами нового стиля, в этой линейке возможности классов нового стиля считаются просто нормальными функциональными средствами классов. Но я решил оставить их описания здесь раздельными из уважения к пользователям кода на Python 2.X — классы в таком коде получают возможности и поведение нового стиля только в случае наследования от `object`.

Другими словами, когда пользователи Python 3.X видят в книге темы, касающиеся "нового стиля", они должны принимать их за описания существующих характеристик своих классов. Для читателей, использующих Python 2.X, они будут набором необяза-

тельных изменений и расширений, которые могут быть включены или нет, если только эксплуатируемый код уже их не задействовал.

В Python 2.X идентифицирующее *синтаксическое* отличие для классов нового стиля связано с тем, что они унаследованы либо от встроенного типа, такого как `list`, либо от особого встроенного класса `object`. Встроенное имя `object` предназначено для того, чтобы служить суперклассом для классов нового стиля, если никакой другой встроенный тип не подходит:

```
class newstyle(object):          # Явное наследование для класса нового
                                 #   стиля в Python 2.X
    ...нормальный код класса...  # Не требуется в Python 3.X:
                                 #   все происходит автоматически
```

Любой класс, производный от `object` или любого другого встроенного типа, автоматически трактуется как класс нового стиля. То есть при условии нахождения встроенного типа где-то в дереве суперклассов класс Python 2.X получает поведение и расширения классов нового стиля. Классы, не являющиеся производными от встроенных типов вроде `object`, считаются классическими.

Что нового в новом стиле?

Как мы увидим, классы нового стиля обладают глубокими отличиями, которые оказывают широкое влияние на программы, особенно когда код задействует добавленные в них расширенные возможности. На самом деле, по крайней мере, с точки зрения их поддержки ООП, эти изменения на ряде уровней превращают Python в *совершенно особый язык*. Они обязательны в линейке Python 3.X, необязательны в линейке Python 2.X, но только если игнорируются каждым программистом, и в данной области заимствуют намного большее из других языков (и часто обладают сравнимой сложностью).

Классы нового стиля частично являются результатом попытки объединить понятие *класса* с понятием *типа* во времена существования версии Python 2.2, хотя для многих они оставались незамеченными, пока не стали необходимым знанием в Python 3.X. Вам нужно самостоятельно оценить успех такого объединения, но как мы выясним, в модели все еще присутствуют различия – теперь между *классом* и *метаклассом* – и один из побочных эффектов заключается в том, что нормальные классы оказываются более мощными, но также и значительно более сложными. Скажем, формализованный в главе 40 алгоритм наследования нового стиля усложнился минимум в два раза.

Тем не менее, некоторые программисты, имеющие дело с прямолинейным прикладным кодом, могут заметить только легкое отклонение от традиционных "классических" классов. Ведь нам же удалось добраться до этого места в книге и попутно реализовать значимые примеры, главным образом просто упоминая о данном изменении. Кроме того, модель классических классов, по-прежнему доступная в Python 2.X, работает в точности так, как работала на протяжении более двух десятилетий.

Однако поскольку классы нового стиля модифицируют основные линии поведения классов, их пришлось вводить в Python 2.X в виде отдельного инструмента, чтобы избежать влияния на любой существующий код, который полагается на предыдущую модель. Например, тонкие отличия, такие как поиск в иерархии наследования с ромбовидными схемами и взаимодействие встроенных операций и методов управляемых атрибутов наподобие `__getattr__`, могут приводить к отказу работы некоторого существующего кода, если оставить его без изменений. Применение необязательных расширений в новой модели вроде слотов может дать аналогичный эффект.

Разделение между моделями классов было устранено в линейке Python 3.X, *предписывающей* использование классов нового стиля, но оно все еще присутствует для читателей, которые применяют Python 2.X или эксплуатируют в производственной среде большой объем кода на Python 2.X. Поскольку классы нового стиля в Python 2.X были необязательным расширением, написанный для данной линейки код мог использовать любую из двух моделей классов.

В следующих двух разделах верхнего уровня приведен обзор отличий классов нового стиля и новых инструментов, которые они предлагают. Рассматриваемые в них темы представляют потенциальные изменения некоторым читателям, работающим с Python 2.X, но просто дополнительные расширенные возможности классов для многих читателей, применяющих Python 3.X. Если вы относитесь ко второй группе, тогда найдете здесь полное описание, несмотря на то, что его часть дается в контексте изменений. Вы вполне можете воспринимать изменения как функциональные средства, но только в случае, если вам никогда не придется иметь дело с любыми из миллионов строк существующего кода на Python 2.X.

Изменения в классах нового стиля

Классы нового стиля отличаются от классических классов в нескольких аспектах, часть которых являются тонкими, но способными повлиять на существующий код на Python 2.X и распространенные стили написания кода. Ниже представлены наиболее заметные отличия как предварительный обзор.

Извлечение атрибутов для встроенных операций: экземпляр пропускается

Обобщенные методы перехвата извлечения атрибутов `__getattr__` и `__getattribute__` по-прежнему выполняются для атрибутов, к которым производится доступ по явному имени, но больше не запускаются для атрибутов, неявно извлекаемых встроенными операциями. Они не вызываются для имен методов перегрузки операций `__X__` только во встроенных контекстах — поиск таких имен начинается в классах, не в экземплярах. Это нарушает работу или усложняет объекты, которые служат в качестве промежуточных для интерфейса другого объекта, если внутренние объекты реализуют перегрузку операций. Методы подобного рода должны быть переопределены из-за отличающегося координирования встроенных операций в классах нового стиля.

Классы и типы объединены: проверка типа

Классы теперь являются типами, а типы — классами. На самом деле по существу они представляют собой синонимы, хотя *метаклассы*, которые теперь относятся к категории типов, все еще кое в чем отличаются от нормальных классов. Встроенная функция `type(I)` возвращает класс, из которого создан экземпляр, а не обобщенный тип экземпляра, и обычно дает такой же разультат, как `I.__class__`. Более того, классы являются экземплярами класса `type`, и можно реализовывать подклассы `type` для настройки создания классов с помощью метаклассов, записываемых посредством операторов `class`. Это может повлиять на код, который проверяет типы или по-другому полагается на предыдущую модель классов.

Автоматический корневой класс `object`: стандартные методы

Все классы нового стиля (отсюда и типы) наследуются от `object`, который содержит небольшой набор стандартных методов перегрузки операций (скажем, `__repr__`). В Python 3.X класс `object` автоматически добавляется выше определяемых пользователем корневых (т.е. самых верхних) классов в дереве и не нуждается в явном указании в качестве суперкласса. Это может повлиять на код, который допускает отсутствие стандартных методов и корневых классов.

Порядок поиска в иерархии наследования: MRO и ромбы

Ромбовидные схемы множественного наследования имеют слегка отличающийся порядок поиска — грубо говоря, в ромбах поиск производится раньше и более в стиле сначала в ширину, чем сначала в глубину. Такой порядок поиска атрибутов, известный как MRO, можно отследить с помощью нового атрибута `__mro__`, доступного в классах нового стиля. Новый порядок поиска в основном применяется только к деревьям классов с ромбами, хотя сам подразумеваемый корень `object` новой модели образует ромб во всех деревьях множественного наследования. Код, который полагается на предыдущий порядок, не будет работать таким же образом.

Алгоритм наследования: глава 40

Алгоритм, используемый при наследовании в классах нового стиля, существенно сложнее, чем модель "сначала в глубину" классических классов, и включает особые случаи для дескрипторов, метаклассов и встроенных операций. Мы не в состоянии формально описать этот алгоритм до главы 40, где будут более подробно рассматриваться метаклассы и дескрипторы, но отметим, что он может оказать воздействие на код, в котором не ожидаются связанные с ним добавочные ухищрения.

Новые расширенные инструменты: влияние на код

Классы нового стиля обладают новыми механизмами, в том числе *слотами*, *свойствами*, *дескрипторами*, встроенной функцией `super` и методом `__getattribute__`. Большинство из них ориентированы на решение весьма специфических задач построения инструментов. Тем не менее, их применение также способно повлиять или нарушить работу существующего кода; например, слоты иногда вообще препятствуют созданию словарей пространств имен экземпляров, а обобщенные обработчики атрибутов могут требовать написания другого кода.

Мы исследуем *расширения*, упомянутые в последнем пункте, в отдельном разделе далее в главе и, как отмечалось выше, отложим раскрытие алгоритма, используемого при наследовании, до главы 40. Однако поскольку другие элементы в приведенном списке потенциально способны нарушить функционирование традиционного кода на Python, давайте пристальнее взглянем на каждый из них по очереди.

Замечание по содержанию. Имейте в виду, что *изменения, введенные в классы нового стиля,* применимы к *обеим* линейкам Python 3.X и 2.X, хотя в Python 2.X они являются лишь возможным вариантом. Как в главе, так и в книге в целом функциональные особенности помечаются как *изменения линейки Python 3.X*, чтобы противопоставить их с традиционным кодом на Python 2.X.

Тем не менее, некоторые из них формально появились в классах нового стиля – они обязательные в Python 3.X, но могут встречаться также в коде на *Python 2.X*. Здесь мы часто указываем на такое различие, но его не следует воспринимать категорично. Усложняя различие, одни изменения, касающиеся классов в Python 3.X, объясняются появлением классов нового стиля (скажем, пропуск `__getattr__` для методов операций), тогда как другие – нет (например, замена несвязанных методов функциями). Кроме того, многие программисты на Python 2.X придерживаются использования классических классов, игнорируя то, что они считают функциональной особенностью Python 3.X. Однако классы нового стиля не новы и применяются в обеих линейках Python – раз уж они появляются в коде на Python 2.X, то обязательны для изучения также пользователями Python 2.X.

Процедура извлечения атрибутов для встроенных операций пропускает экземпляры

Мы представляли это изменение, введенное в классах нового стиля, во врезках в главах 28 и 31 по причине его влияния на предшествующие примеры и темы. В классах нового стиля (и потому во всех классах в Python 3.X) обобщенные методы перехвата атрибутов экземпляров `__getattr__` и `__getattribute__` больше не вызываются встроенными операциями для имен методов перегрузки операций `__X__` – поиск таких имен начинается с классов, а не экземпляров. Тем не менее, доступ к атрибутам по явным именам обрабатывается упомянутыми методами, несмотря на наличие у них имен `__X__`. Следовательно, такое изменение касается главным образом поведения встроенных операций.

Говоря более формально, если в классе определен метод перегрузки операции индексирования `__getitem__` и `X` представляет собой экземпляр данного класса, тогда выражение индексирования вида `X[I]` приблизительно эквивалентно вызову `X.__getitem__(I)` для классических классов, но `type(X).__getitem__(X, I)` для классов нового стиля. Второе выражение начинает свой поиск в классе, поэтому пропускает шаг поиска `__getattr__` в экземпляре для неопределенного имени.

В действительности поиск метода для встроенных операций вроде `X[I]` использует нормальную процедуру наследования, начинающуюся с уровня класса, и инспектирует только словари пространств имен всех классов, от которых наследуется `X`. Такое различие может иметь значение в модели *метаклассов*, которую мы кратко рассмотрим позже в настоящей главе и более подробно в главе 40 – в рамках данной модели классы способны вести себя по-разному. Однако процедура поиска для встроенных операций пропускает экземпляр.

Почему изменился поиск?

Вы можете найти формальные обоснования данного изменения где-то в другом месте; в этой книге нечасто приводятся обстоятельства, объясняющие причину введения того или иного изменения, которое нарушает работу многих существующих программ. Но оно представляется как путь *оптимизации* и решение, казалось бы, неясной проблемы *схемы вызовов*. Первое логическое обоснование подкрепляется частотой применения встроенных операций. Скажем, если каждая операция + требует выполнения дополнительных шагов для экземпляра, то она может уменьшить быстродействие программ – особенно с учетом множества расширений на уровне атрибутов в модели нового стиля.

Второе логическое обоснование менее ясно и описано в руководствах по Python; выражаясь кратко, оно отражает сложную проблему, привнесенную моделью *метаклассов*. Так как классы теперь являются экземплярами метаклассов и поскольку в метаклассах могут определяться методы встроенных операций для обработки классов, генерируемых метаклассами, то вызов метода, запускаемый для класса, обязан пропустить сам класс и произвести поиск на один уровень выше, чтобы подобрать метод, который обработает этот класс, а не выбрать собственную версию метода, принадлежащую классу. Собственная версия привела бы к вызову несвязанного метода, потому что собственный метод класса обрабатывает экземпляры более низкого уровня. Это всего лишь обычная модель несвязанных методов, которая обсуждалась в предыдущей главе, но она потенциально усугубляется тем фактом, что классы способны получать от метаклассов также и поведение типов.

В результате, поскольку классы сами по себе являются и типами, и экземплярами, при поиске методов встроенных операций все экземпляры пропускаются. Такой прием применяется к нормальным экземплярам предположительно ради единообразия и согласованности, но для невстроенных имен, а также прямых и явных обращений к встроенным именам по-прежнему осуществляется проверка экземпляра. Несмотря на то что вероятно это последствие, обусловленное введением модели классов нового стиля, для ряда программистов оно может выглядеть как решение, принятое в пользу менее естественного и более неясного принципа, нежели широко применяемый ранее. Его роль в качестве пути оптимизации кажется в большей степени оправданной, но также не без оговорок.

В частности, изменение поиска оказывает потенциально обширное влияние на классы, основанные на *делегировании*, которые часто называют *классами-посредниками*, когда внедренные объекты реализуют перегрузку операций. В классах нового стиля такой класс-посредник обычно должен *переопределять* любые имена подобного рода для перехвата и делегирования, либо вручную, либо посредством инструментов. Конечным результатом становится либо значительное усложнение, либо полный отказ от *целой категории программ*. Мы исследовали делегирование в главах 28 и 31; оно является распространенной схемой, используемой для дополнения или адаптации интерфейса другого класса — чтобы добавить проверку достоверности, трассировку, измерение времени и многие другие разновидности логики. Хотя в типичном коде на Python классы-посредники могут быть скорее исключением, чем правилом, многие программы полагаются на них.

Последствия для перехвата атрибутов

Выполнив показанное ниже взаимодействие под управлением *Python 2.X*, чтобы посмотреть, чем отличаются классы нового стиля, мы обнаружим, что индексирование и вывод в традиционных классах направляются методу `__getattr__`, но в классах нового стиля вывод использует стандартный метод[1]:

```
>>> class C:
        data = 'spam'
        def __getattr__(self, name):      # Классический класс в Python 2.X:
                                          #   перехватывает встроенные операции
            print(name)
            return getattr(self.data, name)
```

[1] Чтобы сократить размер и уменьшить беспорядок, в листингах взаимодействия в настоящей главе я начал опускать некоторые пустые строки и сокращать шестнадцатеричные адреса объектов до 32 битов. К этому моменту вы наверняка сочтете такие мелкие детали несущественными.

```
>>> X = C()
>>> X[0]
__getitem__
's'
>>> print(X)             # Классический класс не наследует стандартный метод
__str__
spam

>>> class C(object):     # Класс нового стиля в Python 2.X и 3.X
       ...оставшаяся часть класса не изменялась...
>>> X = C()              # Встроенные операции не направляются getattr
>>> X[0]
TypeError: 'C' object does not support indexing
Ошибка типа: объект C не поддерживает индексирование
>>> print(X)
<__main__.C object at 0x02205780>
```

Несмотря на очевидную рационализацию в плане методов метакласса для класса и оптимизации встроенных операций, данное расхождение не касается нормальных экземпляров, имеющих метод `__getattr__`, и применяется только к встроенным операциям – не к нормально именованным методам или явным вызовам встроенных методов по имени:

```
>>> class C: pass        # Классический класс Python 2.X
>>> X = C()
>>> X.normal = lambda: 99
>>> X.normal()
99
>>> X.__add__ = lambda(y): 88 + y
>>> X.__add__(1)
89
>>> X + 1
89
>>> class C(object): pass  # Класс нового стиля Python 2.X/3.X
>>> X = C()
>>> X.normal = lambda: 99
>>> X.normal()         # Нормальные методы по-прежнему поступают из экземпляра
99
>>> X.__add__ = lambda(y): 88 + y
>>> X.__add__(1)         # То же самое для явных встроенных имен
89
>>> X + 1
TypeError: unsupported operand type(s) for +: 'C' and 'int'
Ошибка типа: неподдерживаемые типы операндов для +: C и int
```

Такое поведение наследуется методом перехвата атрибутов `__getattr__`:

```
>>> class C(object):
       def __getattr__(self, name): print(name)
>>> X = C()
>>> X.normal           # Нормальные имена по-прежнему направляются getattr
normal
>>> X.__add__          # Прямые вызовы по имени тоже, но выражения - нет!
__add__
>>> X + 1
TypeError: unsupported operand type(s) for +: 'C' and 'int'
Ошибка типа: неподдерживаемые типы операндов для +: C и int
```

Требования к коду классов-посредников

В более реалистичном сценарии делегирования это значит, что встроенные операции вроде выражений больше не работают таким же образом, как эквивалентные им традиционные прямые вызовы. И наоборот, прямые обращения к именам встроенных методов по-прежнему работают, но эквивалентные выражения — нет, потому что вызовы через тип терпят неудачу для имен не на уровне класса и выше. Другими словами, это различие возникает *только во встроенных операциях*; явные извлечения выполняются корректно:

```
>>> class C(object):
        data = 'spam'
        def __getattr__(self, name):
            print('getattr: ' + name)
            return getattr(self.data, name)

>>> X = C()
>>> X.__getitem__(1)        # Традиционное отображение работает,
                            #  но отображение нового стиля - нет
getattr: __getitem__
'p'

>>> X[1]
TypeError: 'C' object does not support indexing
Ошибка типа: объект C не поддерживает индексирование
>>> type(X).__getitem__(X, 1)
AttributeError: type object 'C' has no attribute '__getitem__'
Ошибка атрибута: объект типа C не имеет атрибута __getitem__

>>> X.__add__('eggs')       # То же самое для +: экземпляр пропускается
                            #  только для выражения
getattr: __add__
'spameggs'

>>> X + 'eggs'
TypeError: unsupported operand type(s) for +: 'C' and 'str'
Ошибка типа: неподдерживаемые типы операндов для +: C и str
>>> type(X).__add__(X, 'eggs')
AttributeError: type object 'C' has no attribute '__add__'
Ошибка атрибута: объект типа C не имеет атрибута __add__
```

Подытожим: при написании кода посредника объекта, к интерфейсу которого частично могут обращаться встроенные операции, классы нового стиля требуют метода `__getattr__` для нормальных имен, а также переопределений методов для всех имен, к которым имеют доступ встроенные операции — кодируются они вручную, получаются из суперклассов или генерируются инструментами. Когда переопределения включаются подобным образом, вызовы через экземпляры *и* через типы эквивалентны встроенным операциям, хотя переопределенные имена больше не направляются обобщенному обработчику неопределенных имен `__getattr__` даже для явных обращений к именам:

```
>>> class C(object):                    # Новый стиль: Python 3.X и 2.X
        data = 'spam'
        def __getattr__(self, name):    # Перехватывать нормальные имена
            print('getattr: ' + name)
            return getattr(self.data, name)
        def __getitem__(self, i):       # Переопределить встроенные операции
            print('getitem: ' + str(i))
```

```
            return self.data[i]                # Выполнить выражение или getattr
        def __add__(self, other):
            print('add: ' + other)
            return getattr(self.data, '__add__')(other)

>>> X = C()
>>> X.upper
getattr: upper
<built-in method upper of str object at 0x0233D670>
>>> X.upper()
getattr: upper
'SPAM'

>>> X[1]                              # Встроенная операция (неявная)
getitem: 1
'p'
>>> X.__getitem__(1)                  # Традиционный эквивалент (явный)
getitem: 1
'p'
>>> type(X).__getitem__(X, 1)         # Эквивалент нового стиля
getitem: 1
'p'

>>> X + 'eggs'                        # То же самое для + и остальных
add: eggs
'spameggs'
>>> X.__add__('eggs')
add: eggs
'spameggs'
>>> type(X).__add__(X, 'eggs')
add: eggs
'spameggs'
```

Дополнительные сведения

Мы возвратимся к обсуждению данного изменения в главе 40, посвященной метаклассам, а также на примере в контекстах управления атрибутами в главе 38 и декораторах защиты доступа в главе 39. В последней из упомянутых глав мы вдобавок исследуем кодовые структуры для обобщенного снабжения посредников обязательными методами операций — вовсе не невозможная задача, которую может понадобиться решить только раз, если сделать это хорошо. Дополнительные сведения о разновидностях кода, затрагиваемых такой проблемой, ищите в более поздних главах и в предшествующих примерах из глав 28 и 31.

Поскольку проблема будет подробно рассматриваться позже в книге, здесь приводится только краткое описание. Ниже приведено несколько рекомендаций.

- ***Рецепты по инструментам.*** Ознакомьтесь с рецептом на Python, доступным по ссылке `http://code.activestate.com/recipes/252151` и описывающим инструмент, который автоматически представляет специальные имена методов как обобщенные диспетчеры вызовов в классе-посреднике, созданном с помощью рассматриваемых далее в главе методик метаклассов. Тем не менее, данный инструмент по-прежнему должен предложить вам передать имена методов операций, которые внутренний объект может реализовать (он обязан, т.к. компоненты интерфейса внутреннего объекта могут быть унаследованы от произвольных источников).

- *Другие подходы.* Поиск в веб-сети в наши дни выявит множество дополнительных инструментов, которые аналогично заполняют классы-посредники методами для перегрузки; это широко распространенная задача! Кроме того, в главе 39 будет показано, каким образом писать код прямолинейных и универсальных суперклассов, которые предоставят требующиеся методы или атрибуты как подмешиваемые, без метаклассов, избыточной генерации кода или подобных сложных методик.

Разумеется, с течением времени история может эволюционировать, но она была проблемой на протяжении многих лет. На сегодняшний день классические классы-посредники для объектов, перегружающих любые операции, фактически не могут работать как классы нового стиля. И в Python 2.X, и в Python 3.X такие классы требуют написания кода или генерации оболочек для всех неявно вызываемых методов операций, которые может поддерживать внутренний объект. Решение не идеально для программ подобного рода (некоторые посредники могут требовать десятков методов оболочки; потенциально свыше 50!), но оно отражает или, по крайней мере, является артефактом целей проектирования у разработчиков классов нового стиля.

Обязательно ознакомьтесь с описанием *метаклассов* в главе 40, чтобы увидеть еще одну демонстрацию проблемы и ее логическое обоснование. Там мы также покажем, что такое поведение встроенных операций квалифицируется как особый случай в *наследовании* нового стиля. Хорошее понимание этого требует большего объема сведений о метаклассах, чем способна предложить текущая глава; заслуживающий сожаления побочный продукт метаклассов в целом связан с тем, что они стали предварительным условием для более широкого употребления, нежели могли предвидеть их создатели.

Изменения модели типов

Перейдем к следующему изменению, привнесенному новым стилем: в зависимости от вашей оценки различие между *типом* и *классом* в классах нового стиля либо значительно сократилось, либо полностью исчезло, как описано ниже.

Классы являются типами

Объект `type` генерирует классы как свои экземпляры, а классы генерируют экземпляры самих себя. Оба считаются типами, потому что они генерируют экземпляры. На самом деле не существует реальной разницы между встроенными типами, такими как списки и строки, и определяемыми пользователем типами, реализованными в виде классов. Вот почему мы можем создавать подклассы встроенных типов, как было показано ранее в главе — подкласс встроенного типа наподобие `list` квалифицируется как класс нового стиля и становится новым типом, определяемым пользователем.

Типы являются классами

Новые типы, генерирующие классы, могут быть реализованы на Python как метаклассы, рассматриваемые позже в главе — подклассы определяемого пользователем типа, которые записываются посредством нормальных операторов `class` и управляют созданием классов, являющихся их экземплярами. Как будет показано, метаклассы являются одновременно классами и типами, хотя они отличаются вполне достаточно, чтобы поддерживать разумный аргумент в пользу того, что предшествующее разветвление "тип/класс" превратилось в "метакласс/класс", возможно ценой добавочной сложности в нормальных классах.

Помимо появления возможности создавать подклассы встроенных типов и реализовывать метаклассы один из самых практических контекстов, где такое объединение "тип/класс" становится наиболее очевидным, касается явной проверки типов. Для классических классов Python 2.X типом экземпляра класса является обобщенный "экземпляр" (instance), но типы встроенных объектов более специфичны:

```
C:\code> c:\python27\python
>>> class C: pass              # Классические классы в Python 2.X

>>> I = C()                    # Экземпляры создаются из классов
>>> type(I), I.__class__
(<type 'instance'>, <class __main__.C at 0x02399768>)

>>> type(C)                    # Но классы не являются тем же, что и типы
<type 'classobj'>
>>> C.__class__
AttributeError: class C has no attribute '__class__'
Ошибка атрибута: класс C не имеет атрибута __class__

>>> type([1, 2, 3]), [1, 2, 3].__class__
(<type 'list'>, <type 'list'>)
>>> type(list), list.__class__
(<type 'type'>, <type 'type'>)
```

Однако для классов нового стиля в Python 2.X типом экземпляра класса будет класс, из которого он создавался, поскольку классы представляют собой просто определяемые пользователем типы – типом экземпляра является его класс, а тип класса, определяемого пользователем, такой же, как тип встроенного объекта. Теперь классы тоже имеют атрибут __class__, потому что они считаются экземплярами type:

```
C:\code> c:\python27\python
>>> class C(object): pass # Классы нового стиля в Python 2.X

>>> I = C()          # Типом экземпляра будет класс, из которого он создавался
>>> type(I), I.__class__
(<class '__main__.C'>, <class '__main__.C'>)

>>> type(C), C.__class__   # Классы являются определяемыми пользователем типами
(<type 'type'>, <type 'type'>)
```

То же самое справедливо для всех классов в Python 3.X, т.к. все классы автоматически относятся к новому стилю, даже если для них явно не указаны суперклассы. Фактически различие между встроенными типами и типами определяемых пользователем классов в Python 3.X, похоже, вообще исчезло:

```
C:\code> c:\python37\python
>>> class C: pass

>>> I = C()            # Все классы в Python 3.X являются классами нового стиля
>>> type(I), I.__class__    # Типом экземпляра будет класс, из которого он создавался
(<class '__main__.C'>, <class '__main__.C'>)

>>> type(C), C.__class__          # Класс - это тип, а тип - это класс
(<class 'type'>, <class 'type'>)

>>> type([1, 2, 3]), [1, 2, 3].__class__
(<class 'list'>, <class 'list'>)

>>> type(list), list.__class__    # Классы и встроенные типы работают одинаково
(<class 'type'>, <class 'type'>)
```

Как видите, в Python 3.X классы – это типы, но типы – также и классы. Формально каждый класс генерируется *метаклассом*, т.е. классом, который обычно является либо самим type, либо подклассом type, настроенным для расширения или управления сгенерированными классами. Кроме влияния на код, который делает проверку типов, это оказывается важной привязкой для разработчиков инструментов. Мы обсудим метаклассы более подробно позже в настоящей главе и еще подробнее в главе 40.

Последствия для проверки типов

Помимо предоставления настройки встроенных типов и привязками метаклассов объединение классов и типов в модели классов нового стиля способно оказывать влияние на код, в котором выполняется проверка типов. Например, в Python 3.X типы экземпляров классов сравниваются напрямую и содержательно, плюс тем же самым способом, что и объекты встроенных типов. Это следует из того факта, что теперь классы являются типами, а типом экземпляра будет класс экземпляра:

```
C:\code> c:\python37\python
>>> class C: pass
>>> class D: pass

>>> c, d = C(), D()
>>> type(c) == type(d)    # Python 3.X: сравниваются классы экземпляров
False

>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)
>>> c1, c2 = C(), C()
>>> type(c1) == type(c2)
True
```

Тем не менее, с классическими классами в Python 2.X сравнение типов экземпляров практически бесполезно, потому что все экземпляры имеют тот же самый тип instance. Чтобы действительно сравнивать типы, потребуется сравнивать атрибуты __class__ экземпляров (если вас заботит переносимость, то такой прием работает и в Python 3.X, но там он необязателен):

```
C:\code> c:\python27\python
>>> class C: pass
>>> class D: pass

>>> c, d = C(), D()
>>> type(c) == type(d)    # Python 2.X: все экземпляры имеют тот же самый тип!
True
>>> c.__class__ == d.__class__      # При необходимости классы можно
                                    # сравнивать явно
False

>>> type(c), type(d)
(<type 'instance'>, <type 'instance'>)
>>> c.__class__, d.__class__
(<class __main__.C at 0x024585A0>, <class __main__.D at 0x024588D0>)
```

И как вы уже к данному моменту должны ожидать, классы нового стиля в Python 2.X работают в этом отношении точно так же, как все классы в Python 3.X – при сравнении типов экземпляров автоматически сравниваются классы экземпляров:

```
C:\code> c:\python27\python
>>> class C(object): pass
>>> class D(object): pass

>>> c, d = C(), D()
>>> type(c) == type(d)      # Классы нового стиля в Python 2.X: так же,
                            #   как все классы в Python 3.X
False

>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)
```

Конечно, как я уже неоднократно отмечал, проверка типов обычно считается неправильным действием в программах на Python (мы пишем код для интерфейсов объектов, а не типов объектов). Более универсальная встроенная функция `isinstance` вероятнее всего будет тем, что вы захотите использовать в тех редких ситуациях, когда должны запрашиваться типы экземпляров классов. Однако знание модели типов Python может помочь пролить свет на модель классов в целом.

Все классы являются производными от object

Еще одно последствие изменения типов в модели классов нового стиля состоит в том, что поскольку все классы являются производными (унаследованными) от класса `object`, либо неявно, либо явно, и по причине того, что теперь все типы — это классы, каждый объект оказывается производным от встроенного класса `object`, будь то напрямую или через суперкласс.

Взгляните на следующее взаимодействие в Python 3.X:

```
>>> class C: pass          # Для классов нового стиля
>>> X = C()
>>> type(X), type(C)       # Тип - это экземпляр класса, из которого он создавался
(<class '__main__.C'>, <class 'type'>)
```

Как и ранее, типом *экземпляра* класса будет класс, из которого он был создан, а типом *класса* — класс `type`, потому что классы и типы объединены. Тем не менее, также верно и то, что экземпляр и класс унаследованы от встроенного класса и типа `object`, т.е. неявного или явного суперкласса каждого класса:

```
>>> isinstance(X, object)
True
>>> isinstance(C, object)    # Классы всегда унаследованы от object
True
```

Предшествующие вызовы `isinstance` в наши дни возвращают одинаковые результаты для классических классов и классов нового стиля в Python 2.X, хотя результаты `type` из Python 2.X отличаются. Как вскоре будет показано, более важно то, что тип `object` не добавляется и потому отсутствует в кортеже `__bases__` классических классов Python 2.X, следовательно, `object` не может считаться подлинным суперклассом.

То же самое отношение остается справедливым для встроенных типов вроде списков и строк, т.к. в модели нового стиля типы являются классами — встроенные типы теперь стали классами и их экземпляры тоже унаследованы от `object`:

```
>>> type('spam'), type(str)
(<class 'str'>, <class 'type'>)
```

```
>>> isinstance('spam', object)  # То же самое для встроенных типов (классов)
True
>>> isinstance(str, object)
True
```

В действительности сам type унаследован от object, а object от type, хотя они представляют собой разные объекты — циклическое отношение, которое завершает объектную модель и происходит из того факта, что типы являются классами, генерирующими классы:

```
>>> type(type)                  # Все классы - это типы и наоборот
<class 'type'>
>>> type(object)
<class 'type'>

>>> isinstance(type, object)    # Все классы являются производными от object,
                                #   даже type
True
>>> isinstance(object, type)    # Типы создают классы и type - это класс
True
>>> type is object
False
```

Последствия для стандартных методов

Приведенные выше сведения могут показаться непонятными, но с этой моделью связано несколько практических последствий. Прежде всего, временами мы должны знать о стандартных методах, которые поступают из явного или неявного корневого класса object в классах нового стиля:

```
c:\code> py -2
>>> dir(object)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__',
'__hash__', '__init__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']

>>> class C: pass
>>> C.__bases__                # Классические классы не унаследованы от object
()
>>> X = C()
>>> X.__repr__
AttributeError: C instance has no attribute '__repr__'
AttributeError: экземпляр C не имеет атрибута __repr__

>>> class C(object): pass      # Классы нового стиля наследуют стандартные
                               #   методы object
>>> C.__bases__
(<type 'object'>,)
>>> X = C()
>>> X.__repr__
<method-wrapper '__repr__' of C object at 0x00000000020B5978>

c:\code> py -3
>>> class C: pass              # Это означает, что в Python 3.X все классы
                               #   получают стандартные методы
>>> C.__bases__
(<class 'object'>,)
>>> C().__repr__
<method-wrapper '__repr__' of C object at 0x0000000002955630>
```

Модель классов нового стиля также учитывает меньше особых случаев, чем бывшее в модели классических классов различие между типом и классом, и позволяет нам писать код, который безопасно предполагает наличие и пользуется суперклассом `object` (скажем, принимая его в качестве "привязки" в рассматриваемых далее ролях встроенной функции `super` и передавая ему вызовы методов для запуска стандартной линии поведения). Мы продемонстрируем примеры с участием `super` позже в книге, а пока давайте займемся исследованием последнего значительного изменения в модели классов нового стиля.

Изменение ромбовидного наследования

Финальное изменение в модели классов нового стиля также является одним из самых заметных: слегка отличающийся порядок поиска для так называемых деревьев множественного наследования с *ромбовидными* схемами. Наличие в таких деревьях более чем одного суперкласса приводит к тому же самому расположенному выше суперклассу (их название происходит от ромбовидной формы дерева, если вы его нарисуете — квадрат, опирающийся на один из его углов).

Ромбовидная схема — довольно сложная концепция проектирования, которая встречается только в деревьях множественного наследования и обычно редко применяется в практике программирования на Python, поэтому мы не будем раскрывать данную тему особенно глубоко. Однако отличающиеся порядки поиска были кратко представлены при рассмотрении множественного наследования в предыдущей главе.

Для классических классов (стандарт в Python 2.X): DFLR

Путь поиска при наследовании проходит строго сначала в глубину и затем слева направо — Python поднимается все время к верху, придерживаясь левой стороны дерева, и только потом останавливается и начинает просмотр дальше вправо. Такой порядок поиска известен как *DFLR* (Depth-First, Left-to-Right — сначала в глубину, слева направо).

Для классов нового стиля (необязательные в Python 2.X и автоматические в Python 3.X): MRO

Путь поиска при наследовании в ромбовидных схемах выполняется больше в манере сначала в ширину — Python сначала ищет в любых суперклассах справа от только что просмотренного и только потом поднимается к общему суперклассу вверху. Другими словами, поиск проходит по уровням, прежде чем двигаться вверх. Такой порядок поиска называется *MRO* нового стиля (Method Resolution Order — порядок распознавания методов), а часто ради краткости — просто MRO, когда используется для противопоставления с порядком DFLR. Несмотря на свое название, он применяется для всех атрибутов в Python, а не только для методов.

Алгоритм MRO нового стиля немного сложнее, чем было представлено выше (и позже мы опишем его более формально), но именно столько нужно знать многим программистам. Тем не менее, важно отметить, что он обладает не только важными преимуществами для кода с классами нового стиля, но и потенциалом нарушения работы существующего кода с классическими классами.

Например, алгоритм MRO нового стиля дает возможность нижним суперклассам перегружать атрибуты верхних суперклассов, не обращая внимания на разновидности деревьев множественного наследования, в которых они смешаны. Кроме того правило поиска нового стиля позволяет избежать посещения того же самого суперкласса

более одного раза, когда он доступен из множества подклассов. Вероятно алгоритм MRO лучше DFLR, но он применяется к небольшому подмножеству пользовательского кода на Python; однако, как мы увидим, модель классов нового стиля *сама по себе* делает ромбы гораздо более распространенными, а MRO более важным.

В то же самое время новый алгоритм MRO будет определять местонахождение атрибутов по-другому, создавая потенциальную несовместимость для классических классов Python 2.X. Давайте перейдем к исследованию какого-нибудь кода, чтобы посмотреть, как отличия проявляются на практике.

Последствия для деревьев ромбовидного наследования

Для иллюстрации отличий поиска по алгоритму MRO нового стиля рассмотрим упрощенный пример множественного наследования с ромбовидной схемой для *классических классов*. Здесь суперклассы B и C класса D ведут к тому же самому общему предку A:

```
>>> class A: attr = 1          # Классический класс (Python 2.X)
>>> class B(A): pass           # B и C ведут к A
>>> class C(A): attr = 2
>>> class D(B, C): pass        # Проверяет A перед C

>>> x = D()
>>> x.attr                     # Ищет в x, D, B, A
1
```

Атрибут x.attr обнаруживается в суперклассе A, потому что в классических классах поиск при наследовании поднимается настолько высоко, насколько может, прежде чем остановиться и начать движение вправо. Полный порядок поиска DFLR посетит x, D, B, A, C и затем A. Для указанного атрибута поиск прекращается, как только attr встречается в A, выше B.

Тем не менее, в *классах нового стиля*, производных от встроенного типа вроде object (и во всех классах Python 3.X), порядок поиска отличается: Python просматривает C справа от B до проверки A выше B. Полный порядок поиска MRO посетит x, D, B, C и затем A. Для атрибута x.attr поиск прекращается, когда attr встречается в C:

```
>>> class A(object): attr = 1  # Классы нового стиля
                               # (в Python 3.X указывать object не требуется)
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B, C): pass        # Проверяет C перед A

>>> x = D()
>>> x.attr                     # Ищет в x, D, B, C
2
```

Изменение в процедуре поиска при наследовании основано на допущении о том, что если вы подмешиваете класс C ниже в дереве, то вероятно намереваетесь захватить его атрибуты, отдавая им предпочтение перед атрибутами из A. Также допускалось, что класс C всегда должен переопределять атрибуты A во всех контекстах. Скорее всего, так и будет, если класс C используется автономно, но может не произойти, когда он подмешивается в ромбовидную схему с классическими классами — при написании кода класса об этом можно даже не подозревать.

Однако поскольку наиболее вероятно, что программист имел в виду как раз переопределение классом C атрибутов A в подобной ситуации, классы нового стиля посещают класс C первым. Иначе в ромбовидном контексте класс C мог бы стать по существу бесполезным для любых имен в A — настройка A оказалась бы невозможной, и применялись бы только имена, уникальные для C.

Явное устранение конфликтов

Разумеется, проблема с допущениями в том, что они что-то предполагают! Если такое расхождение порядка поиска кажется слишком тонким, чтобы помнить о нем, или вам необходим более полный контроль над процессом поиска, то вы всегда можете принудительно выбрать атрибут откуда угодно в дереве, выполнив присваивание или по-другому указав желаемый атрибут в месте, где классы смешиваются. Скажем, ниже демонстрируется выбор порядка нового стиля в классическом классе за счет явного выбора:

```
>>> class A: attr = 1                 # Классический класс
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B, C): attr = C.attr      # <== Выбор C, справа

>>> x = D()
>>> x.attr                            # Работает подобно классам нового стиля
                                      #   (всем классам в Python 3.X)
2
```

Здесь дерево классических классов эмулирует порядок поиска, принятый в классах нового стиля, для специфического атрибута: присваивание значения атрибуту в `D` выбирает версию в `C`, нарушая тем самым нормальный путь поиска при наследовании (`D.attr` будет самым нижним в дереве). Классы нового стиля могут аналогичным образом эмулировать классические классы за счет выбора более высокой версии целевого атрибута в месте, где они смешиваются:

```
>>> class A(object): attr = 1         # Классы нового стиля
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B, C): attr = B.attr      # <== Выбор A.attr, выше

>>> x = D()
>>> x.attr                            # Работает подобно классическим классам
                                      #   (стандарт в Python 2.X)
1
```

Если вы готовы всегда устранять конфликты подобным образом, то можете почти полностью проигнорировать различие в порядке поиска и не полагаться на допущения о том, что вы имели в виду при написании кода своих классов.

Естественно, выбираемые таким способом атрибуты также могут быть функциями методов — методы являются нормальными поддерживающими присваивание атрибутами, которые предназначены для ссылки на вызываемые объекты функций:

```
>>> class A:
        def meth(s): print('A.meth')

>>> class C(A):
        def meth(s): print('C.meth')

>>> class B(A):
        pass

>>> class D(B, C): pass    # Использовать стандартный порядок поиска
>>> x = D()                # Будет варьироваться в зависимости от типа класса
>>> x.meth()               # Стандартный классический порядок в Python 2.X
A.meth

>>> class D(B, C): meth = C.meth      # <== Выбирает метод из C: новый стиль
                                      #   (и Python 3.X)
```

```
>>> x = D()
>>> x.meth()
C.meth

>>> class D(B, C): meth = B.meth   # <== Выбирает метод из B: классический
>>> x = D()
>>> x.meth()
A.meth
```

Мы выбираем методы, явно присваивая значения именам ниже в дереве. Мы могли бы также просто явно обращаться к желаемому классу; на практике показанная схема может оказаться более распространенной, особенно для таких вещей, как конструкторы:

```
class D(B, C):
    def meth(self):                  # Переопределить нижний
        ...
        C.meth(self)                 # <== Выбирает метод из C по вызову
```

Такой выбор по присваиванию или обращению в точках смешивания может эффективно изолировать ваш код от различия в вариантах классов. Конечно, прием применим только к атрибутам, поддерживаемым подобным образом, но явное устранение конфликтов гарантирует, что ваш код не будет варьироваться от версии к версии Python, во всяком случае, с точки зрения выбора при конфликте атрибутов. Другими словами, это способно служить методикой *переносимости* для классов, которые может потребоваться запускать в рамках моделей классов нового стиля и классических классов.

Явное лучше неявного – для распознавания методов тоже. Даже без расхождения между классическими классами и классами нового стиля показанная здесь методика явного распознавания методов в целом может пригодиться во многих сценариях наследования. Например, если вам нужна часть суперкласса слева и часть суперкласса справа, тогда вам может понадобиться сообщить Python, какие одинаково именованные атрибуты выбирать, за счет использования в подклассах явных присваиваний или обращений. Мы еще возвратимся к этой идее при рассмотрении затруднений в конце главы.

Также обратите внимание, что наследование с ромбовидными схемами в ряде случаев может оказаться более проблематичным, чем вытекает из обсуждения выше (скажем, что если B и C оба требуют конструкторов, которые вызывают конструктор в A?). Поскольку при реальной разработке на Python такие контексты встречаются редко, мы рассмотрим эту тему после исследования встроенной функции super ближе к концу главы. Кроме предоставления обобщенного доступа суперклассам в деревьях одиночного наследования встроенная функция super поддерживает кооперативный режим для устранения конфликтов в деревьях множественного наследования путем упорядочения вызовов методов в соответствии с MRO – при условии, что такой порядок имеет смысл в данном контексте!

Пределы влияния изменения в порядке поиска

Подводя итоги, по умолчанию поиск в ромбовидной схеме для классических классов и классов нового стиля выполняется по-разному, и это изменение не является обратно совместимым. Тем не менее, имейте в виду, что данное изменение влияет главным образом на случаи множественного наследования с ромбовидными схемами; поиск при наследовании с классами нового стиля работает одинаково для большинства других структур деревьев наследования. Вдобавок не исключено, что вся проблема

может иметь скорее теоретическое, нежели практическое значение. Поскольку поиск нового стиля не был достаточно важным для решения до версии Python 2.2 и не стал стандартом вплоть до версии Python 3.0, то он вряд ли повлияет на большинство кода на Python.

После сказанного я также должен отметить, что даже если вы не будете применять ромбовидные схемы в собственных классах, из-за наличия подразумеваемого суперкласса `object` выше любого корневого класса в Python 3.X на сегодняшний день *каждый* случай множественного наследования демонстрирует ромбовидную схему. То есть в классах нового стиля `object` автоматически исполняет ту же роль, которую исполнял класс `A` в рассмотренном ранее примере. Следовательно, правило поиска MRO нового стиля не только модифицирует логическую семантику, но также представляет собой важную *оптимизацию производительности* – оно позволяет избежать посещения и поиска в отдельно взятом классе более одного раза, даже в автоматически добавляемом классе `object`.

Не менее важен и тот факт, что подразумеваемый суперкласс `object` в модели классов нового стиля предоставляет *стандартные методы* для разнообразных встроенных операций, включая методы форматов отображения `__str__` и `__repr__`. Запустите `dir(object)`, чтобы получить перечень доступных методов. Без правила поиска MRO нового стиля в сценариях с множественным наследованием стандартные методы в `object` всегда замещали бы переопределения в пользовательских классах, если только переопределения не располагались бы в крайнем слева суперклассе. Другими словами, сама модель классов нового стиля делает использование порядка поиска нового стиля более важным!

Чтобы ознакомиться с наглядным примером работы с подразумеваемым суперклассом `object` в Python 3.X и другими примерами создаваемых им ромбовидных схем, просмотрите вывод класса `ListTree` из файла `lister.py`, представленного в предыдущей главе, а также код примера обхода деревьев `classtree.py` в главе 29 – равно как и материал следующего раздела.

Дополнительные сведения о MRO: порядок распознавания методов

Чтобы отследить, каким образом наследование нового стиля работает по умолчанию, мы также можем применять новый атрибут *класс*.`__mro__`, упоминаемый при рассмотрении примера `lister.py` в предыдущей главе – формально расширение нового стиля, но полезное при исследовании изменения. Этот атрибут возвращает *MRO* класса – порядок, в котором процедура наследования осуществляет поиск в классах в дереве классов нового стиля. Реализация MRO основана на алгоритме линеаризации суперклассов C3, первоначально разработанного в языке программирования Dylan, но позже принятого другими языками, в том числе Python 2.3 и Perl 6.

Алгоритм MRO

Полное описание алгоритма MRO в книге не приводится намеренно, поскольку многим программистам на Python знать его попросту не нужно (MRO касается только ромбовидных схем, относительно редко встречающихся в реальном коде), он отличается между линейками Python 2.X и Python 3.X, а детали MRO чересчур загадочны и академичны для настоящей книги. Предпочтение в книге, как правило, отдается неформальному изучению алгоритмов на примерах.

С другой стороны, некоторых читателей может интересовать формальная теория, лежащая в основе MRO нового стиля. Если вы относитесь к ним, тогда поищите полное описание в руководствах по Python и в веб-сети. Однако ниже кратко изложена суть работы MRO.

1. Построение списка всех классов, от которых унаследован экземпляр, с использованием правила поиска DFLR классических классов и многократное включение класса, если он посещается более одного раза.
2. Просмотр построенного списка на предмет дубликатов с удалением всех вхождений кроме последнего.

Результирующий список MRO для заданного класса включает этот класс, его суперклассы и все более высокие суперклассы вплоть до корневого класса `object`, расположенного на верхушке дерева. Он упорядочен так, что каждый класс находится перед своими родителями, а множество родителей сохраняют порядок, в котором они следуют внутри кортежа суперклассов `__bases__`.

Тем не менее, важно отметить, что поскольку общие родители в *ромбах* появляются только в позициях, в которых они посещались *последний* раз, поиск в нижних классах выполняется первым, когда позже список MRO применяется при наследовании атрибутов. Кроме того, каждый класс включается и потому посещается только один раз независимо от того, сколько классов к нему ведет.

Практические приложения данного алгоритма будут представлены далее в главе, включая его использование в `super` — встроенной функции, применение которой делает обязательным изучение MRO, если вы хотите полностью разобраться в том, каким образом координируются методы в случае вызова этой функции. Как мы увидим, вопреки своему имени вызов `super` порождает обращение к следующему классу в списке MRO, который вообще может не быть суперклассом.

Отслеживание MRO

На тот случай, когда вы просто хотите посмотреть, каким образом процедура наследования нового стиля упорядочивает классы в целом, классы нового стиля (т.е. все классы в Python 3.X) имеют атрибут *`класс`*`.__mro__`. Указанный атрибут представляет собой кортеж, который дает линейный порядок поиска, используемый Python при просмотре атрибутов в суперклассах. В действительности этот атрибут *является* порядком наследования в классах нового стиля и часто оказывается единственной деталью, относящей к MRO, знание которой вполне достаточно для многих пользователей Python.

Ниже приведено несколько иллюстративных примеров, запускаемых под управлением Python 3.X. В сценариях наследования с *ромбовидными* схемами при поиске применяется ранее исследованный новый порядок — *в ширину* перед подъемом согласно MRO для классов нового стиля, всегда используемых в Python 3.X и доступных в качестве варианта в Python 2.X:

```
>>> class A: pass
>>> class B(A): pass    # Ромбы: для классов нового стиля порядок отличается
>>> class C(A): pass    # Поиск сначала в ширину на нижних уровнях
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

Однако для *неромбовидных* схем поиск ведет себя так, как было всегда (хотя и с дополнительным корнем object) – вверх и затем вправо (порядок поиска *DFLR*, применяемый классическими классами в Python 2.X):

```
>>> class A: pass
>>> class B(A): pass       # Неромбовидные схемы: порядок такой же,
                           #   как у классических классов
>>> class C: pass          # Сначала в глубину, затем слева направо
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>,
<class '__main__.C'>, <class 'object'>)
```

Например, порядок MRO следующего дерева будет таким же, как у показанного ранее ромба, согласно DFLR:

```
>>> class A: pass
>>> class B: pass          # Еще одна неромбовидная схема: DFLR
>>> class C(A): pass
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

Обратите внимание на то, что подразумеваемый суперкласс object всегда появляется в *конце* MRO; как уже упоминалось, он добавляется автоматически выше корневых (*самых верхних*) классов в деревьях классов нового стиля в Python 3.X (и необязательно в Python 2.X):

```
>>> A.__bases__           # Связи между суперклассами: object выше двух корней
(<class 'object'>,)
>>> B.__bases__
(<class 'object'>,)
>>> C.__bases__
(<class '__main__.A'>,)
>>> D.__bases__
(<class '__main__.B'>, <class '__main__.C'>)
```

Формально подразумеваемый суперкласс object всегда образует ромб в дереве множественного наследования, даже когда ваши классы этого не делают – поиск в ваших классах выполняется, как и ранее, но MRO нового стиля гарантирует, что object посещается последним, так что ваши классы могут переопределять его стандартные методы:

```
>>> class X: pass
>>> class Y: pass
>>> class A(X): pass       # Неромбовидная схема: сначала в глубину,
                           #   затем слева направо
>>> class B(Y): pass       # Тем не менее, подразумеваемый суперкласс object
                           #   всегда образует ромб
>>> class D(A, B): pass
>>> D.mro()
[<class '__main__.D'>, <class '__main__.A'>, <class '__main__.X'>,
<class '__main__.B'>, <class '__main__.Y'>, <class 'object'>]
>>> X.__bases__, Y.__bases__
((<class 'object'>,), (<class 'object'>,))
>>> A.__bases__, B.__bases__
((<class '__main__.X'>,), (<class '__main__.Y'>,))
```

Атрибут *класс*.`__mro__` доступен лишь в классах нового стиля; в классах Python 2.X он отсутствует, если только они не унаследованы от `object`. Строго говоря, классы нового стиля также имеют метод *класс*.`mro()`, который ради разнообразия использовался в предыдущем примере; он вызывается на стадии создания класса и возвращает список, применяемый для инициализации атрибута `__mro__`, когда класс создан (данный метод доступен для настройки в описываемых позже метаклассах). Вы также можете выбирать имена MRO, если отображение объектов классов оказывается излишне подробным, хотя в книге обычно показываются *объекты*, чтобы напоминать вам об их подлинной форме:

```
>>> D.mro() == list(D.__mro__)
True
>>> [cls.__name__ for cls in D.__mro__]
['D', 'A', 'X', 'B', 'Y', 'object']
```

Тем не менее, при их доступе или отображении пути MRO классов могут быть полезны в плане устранения путаницы, а также в инструментах, которые должны имитировать порядок поиска при наследовании в Python. В следующем разделе последняя роль демонстрируются в действии.

Пример: отображение атрибутов на источники наследования

В качестве основного сценария использования MRO в конце предыдущей главы мы отметили, что инструменты подъема по деревьям классов (наподобие реализованного примера `lister.py`) могли бы извлечь преимущества из MRO. В том виде инструмент вывода списка давал *физические* местоположения атрибутов в дереве классов. Однако за счет отображения списка унаследованных атрибутов из результата `dir` на линейную последовательность MRO (или порядок DFLR для классических классов) такие инструменты могут более прямо ассоциировать атрибуты с классами, от которых они *унаследованы* – в равной мере полезная взаимосвязь для программистов.

Здесь мы перепишем наш инструмент вывода списка, но сначала предпримем первый крупный шаг, реализовав в файле `mapattrs.py` инструменты, которые можно применять для ассоциирования атрибутов с их источниками наследования. Вдобавок его функция `mapattrs` продемонстрирует, каким образом процедура наследования фактически ищет атрибуты в объектах дерева классов, хотя MRO нового стиля в значительной степени автоматизирует процесс:

```
"""
Файл mapattrs.py (Python 3.X + 2.X)

Главный инструмент: mapattrs() отображает все собственные
и унаследованные атрибуты экземпляра на экземпляр или класс,
из которого они унаследованы.

Предполагает, что dir() выдает все атрибуты экземпляра.
Для эмуляции наследования использует либо кортеж MRO класса,
который дает порядок поиска для классов нового стиля
(и всех классов в Python 3.X), либо рекурсивный обход для
выведения порядка DFLR классических классов в Python 2.X.

Также здесь: inheritance() дает нейтральный к версии порядок следования
классов;
различные словарные инструменты, использующие включения Python 3.X/2.7.
"""
```

```
import pprint
def trace(X, label='', end='\n'):
    print(label + pprint.pformat(X) + end)    # Симпатичный вывод

def filterdictvals(D, V):
    """
    Словарь D с элементами после удаления значения V.
    filterdictvals(dict(a=1, b=2, c=1), 1) => {'b': 2}
    """
    return {K: V2 for (K, V2) in D.items() if V2 != V}

def invertdict(D):
    """
    Словарь D со значениями, измененными на ключи (сгруппированными по значениям).
    Все значения должны быть хешируемыми, чтобы работать
    как ключи словаря/множества.
    invertdict(dict(a=1, b=2, c=1)) => {1: ['a', 'c'], 2: ['b']}
    """
    def keysof(V):
        return sorted(K for K in D.keys() if D[K] == V)
    return {V: keysof(V) for V in set(D.values())}

def dflr(cls):
    """
    Классический порядок сначала в глубину, затем слева направо
    в дереве классов cls.
    Циклы невозможны: Python запрещает изменения в __bases__.
    """
    here = [cls]
    for sup in cls.__bases__:
        here += dflr(sup)
    return here

def inheritance(instance):
    """
    Порядок поиска при наследовании: нового стиля (MRO) или классический (DFLR)
    """
    if hasattr(instance.__class__, '__mro__'):
        return (instance,) + instance.__class__.__mro__
    else:
        return [instance] + dflr(instance.__class__)

def mapattrs(instance, withobject=False, bysource=False):
    """
    Словарь с ключами, дающими все унаследованные атрибуты экземпляра,
    и значениями, содержащими объекты, от которых унаследован каждый атрибут.
    withobject: False=удалить встроенные атрибуты класса object.
    bysource: True=сгруппировать результат по объектам, а не атрибутам.
    Поддерживает классы со слотами, которые препятствуют наличию __dict__
    в экземплярах.
    """
    attr2obj = {}
    inherits = inheritance(instance)
    for attr in dir(instance):
        for obj in inherits:
            if hasattr(obj, '__dict__') and attr in obj.__dict__:    # Слоты
                attr2obj[attr] = obj
                break
```

```
    if not withobject:
        attr2obj = filterdictvals(attr2obj, object)
    return attr2obj if not bysource else invertdict(attr2obj)
if __name__ == '__main__':
    print('Classic classes in 2.X, new-style in 3.X') # Классические классы
                                                      #   в Python 2.X,
    # классы нового стиля в Python 3.X
    class A: attr1 = 1
    class B(A): attr2 = 2
    class C(A): attr1 = 3
    class D(B, C): pass
    I = D()
    print('Py=>%s' % I.attr1)               # Совпадает ли поиск Python с нашим?
    trace(inheritance(I), 'INH\n')          # [Порядок наследования]
    trace(mapattrs(I), 'ATTRS\n')           # Атрибуты => Источник
    trace(mapattrs(I, bysource=True), 'OBJS\n')    # Источник => [Атрибуты]
    print('New-style classes in 2.X and 3.X') # Классы нового стиля в 2.X и 3.X
    class A(object): attr1 = 1                # В Python 3.X указывать
                                              #   (object) необязательно
    class B(A): attr2 = 2
    class C(A): attr1 = 3
    class D(B, C): pass
    I = D()
    print('Py=>%s' % I.attr1)
    trace(inheritance(I), 'INH\n')
    trace(mapattrs(I), 'ATTRS\n')
    trace(mapattrs(I, bysource=True), 'OBJS\n')
```

В файле mapattrs.py предполагается, что dir дает все атрибуты экземпляра. Каждый атрибут из результата dir отображается на его источник за счет просмотра дерева либо в порядке MRO для классов нового стиля, либо в порядке DFLR для классических классов и попутно производится поиск в словаре `__dict__` каждого объекта. Для классических классов порядок DFLR строится с помощью простого рекурсивного просмотра. Совокупным эффектом будет эмуляция поиска при наследовании, выполняемого Python для обеих моделей классов.

Код самотестирования этого файла применяет его инструменты к деревьям множественного наследования с ромбовидными схемами, которые мы видели ранее. Для симпатичного отображения списков и словарей он использует библиотечный модуль Python по имени pprint — базовый вызов выглядит как pprint.pprint, а pformat возвращает строку вывода. Запустите код под управлением Python 2.7, чтобы увидеть порядки поиска DFLR и MRO; в Python 3.7 наследовать от object необязательно, и оба теста дают те же самые результаты нового стиля. Важно отметить, что attr1, значение которого помечено посредством Py=>, а имя появляется в списках результатов, наследуется из класса A при классическом поиске, но из класса C при поиске нового стиля:

```
c:\code> py -2 mapattrs.py
Classic classes in 2.X, new-style in 3.X
Py=>1
INH
[<__main__.D instance at 0x000000000225A688>,
 <class __main__.D at 0x0000000002248828>,
 <class __main__.B at 0x0000000002248768>,
 <class __main__.A at 0x0000000002248708>,
 <class __main__.C at 0x00000000022487C8>,
 <class __main__.A at 0x0000000002248708>]
```

```
ATTRS
{'__doc__': <class __main__.D at 0x0000000002248828>,
 '__module__': <class __main__.D at 0x0000000002248828>,
 'attr1': <class __main__.A at 0x0000000002248708>,
 'attr2': <class __main__.B at 0x0000000002248768>}
OBJS
{<class __main__.A at 0x0000000002248708>: ['attr1'],
 <class __main__.B at 0x0000000002248768>: ['attr2'],
 <class __main__.D at 0x0000000002248828>: ['__doc__', '__module__']}
New-style classes in 2.X and 3.X
Py=>3
INH
(<__main__.D object at 0x0000000002257B38>,
 <class '__main__.D'>,
 <class '__main__.B'>,
 <class '__main__.C'>,
 <class '__main__.A'>,
 <type 'object'>)
ATTRS
{'__dict__': <class '__main__.A'>,
 '__doc__': <class '__main__.D'>,
 '__module__': <class '__main__.D'>,
 '__weakref__': <class '__main__.A'>,
 'attr1': <class '__main__.C'>,
 'attr2': <class '__main__.B'>}
OBJS
{<class '__main__.A'>: ['__dict__', '__weakref__'],
 <class '__main__.B'>: ['attr2'],
 <class '__main__.C'>: ['attr1'],
 <class '__main__.D'>: ['__doc__', '__module__']}
```

В качестве более крупного приложения созданных инструментов ниже демонстрируется работа нашего эмулятора наследования в Python 3.7 для рассмотренных в предыдущей главе тестовых классов из файла `testmixin0.py` (ради экономии места некоторые встроенные имена не показаны; запустите код самостоятельно, чтобы получить полный список). Обратите внимание на то, что псевдозакрытые имена `__X` отображаются на классы, где они определены, а `ListInstance` появляется в MRO *перед* классом `object`, имеющим метод `__str__`, который иначе был бы выбран первым – как вы помните, подмешивание этого метода было основной целью классов вывода списков!

```
c:\code> py -3
>>> from mapattrs import trace, dflr, inheritance, mapattrs
>>> from testmixin0 import Sub
>>> I = Sub()            # Класс Sub унаследован от корней Super и ListInstance
>>> trace(dflr(I.__class__))     # Порядок поиска Python 2.X: подразумеваемый
                                 #   object перед ListInstance!
[<class 'testmixin0.Sub'>,
 <class 'testmixin0.Super'>,
 <class 'object'>,
 <class 'listinstance.ListInstance'>,
 <class 'object'>]
>>> trace(inheritance(I))       # Порядок поиска Python 3.X (+ новый стиль
                                #   Python 2.X): ListInstance первый
```

```
(<testmixin0.Sub object at 0x0000000002974630>,
 <class 'testmixin0.Sub'>,
 <class 'testmixin0.Super'>,
 <class 'listinstance.ListInstance'>,
 <class 'object'>)
>>> trace(mapattrs(I))
{'_ListInstance__attrnames': <class 'listinstance.ListInstance'>,
 '__dict__': <class 'testmixin0.Super'>,
 '__doc__': <class 'testmixin0.Sub'>,
 '__init__': <class 'testmixin0.Sub'>,
 '__module__': <class 'testmixin0.Sub'>,
 '__str__': <class 'listinstance.ListInstance'>,
 '__weakref__': <class 'testmixin0.Super'>,
 'data1': <testmixin0.Sub object at 0x02DAEA90>,
 'data2': <testmixin0.Sub object at 0x02DAEA90>,
 'data3': <testmixin0.Sub object at 0x02DAEA90>,
 'ham': <class 'testmixin0.Super'>,
 'spam': <class 'testmixin0.Sub'>}
>>> trace(mapattrs(I, bysource=True))
{<testmixin0.Sub object at 0x02DAEA90>: ['data1', 'data2', 'data3'],
 <class 'listinstance.ListInstance'>: ['_ListInstance__attrnames', '__str__'],
 <class 'testmixin0.Super'>: ['__dict__', '__weakref__', 'ham'],
 <class 'testmixin0.Sub'>: ['__doc__', '__init__', '__module__', 'spam']}
>>> trace(mapattrs(I, withobject=True))
{'_ListInstance__attrnames': <class 'listinstance.ListInstance'>,
 '__class__': <class 'object'>,
 '__delattr__': <class 'object'>,
 ...и так далее...
```

Вот код, который вы можете запустить, если хотите добавить к объектам классов имена, унаследованные экземпляром. Правда, имеет смысл отбросить некоторые встроенные имена с двумя символами подчеркивания, просто чтобы сохранить зрение пользователям!

```
>>> amap = mapattrs(I, withobject=True, bysource=True)
>>> trace(amap)
{<testmixin0.Sub object at 0x02DAEA90>: ['data1', 'data2', 'data3'],
 <class 'listinstance.ListInstance'>: ['_ListInstance__attrnames', '__str__'],
 <class 'testmixin0.Super'>: ['__dict__', '__weakref__', 'ham'],
 <class 'testmixin0.Sub'>: ['__doc__', '__init__', '__module__', 'spam'],
 <class 'object'>: ['__class__',
                    '__delattr__',
                     ...и так далее...
                    '__sizeof__',
                    '__subclasshook__']}
```

В заключение, также следуя размышлениям из предыдущей главы и плавно переходя к следующему разделу, ниже показано, как данная схема работает для *слотовых* атрибутов, основанных на классах. Поскольку словарь `__dict__` класса включает нормальные атрибуты класса и отдельные элементы для атрибутов экземпляров, определяемых списком `__slots__`, слотовые атрибуты, унаследованные экземпляром, будут корректно ассоциированы с реализующим классом, из которого они получены, даже если они физически не хранятся в самом словаре `__dict__` экземпляра:

```
# mapattrs-slots.py: тестирует наследование атрибутов __slots__
from mapattrs import mapattrs, trace

class A(object): __slots__ = ['a', 'b']; x = 1; y = 2
class B(A): __slots__ = ['b', 'c']
class C(A): x = 2
class D(B, C):
    z = 3
    def __init__(self): self.name = 'Bob';

I = D()
trace(mapattrs(I, bysource=True))    # Также trace(mapattrs(I))
```

Для однозначных классов нового стиля вроде представленных в файле `mapattrs-slots.py`, результаты одинаковы в Python 2.7 и 3.7, хотя Python 3.7 добавляет дополнительные встроенные имена. Имена атрибутов здесь воспроизводят все атрибуты, унаследованные экземпляром из определяемых пользователем классов, несмотря на то, что они реализованы слотами, которые определены на уровне классов и хранятся в пространстве, выделенном внутри экземпляра:

```
c:\code> py -3 mapattrs-slots.py
{<__main__.D object at 0x02DC5450>: ['name'],
 <class '__main__.A'>: ['a', 'y'],
 <class '__main__.B'>: ['__slots__', 'b', 'c'],
 <class '__main__.C'>: ['x'],
 <class '__main__.D'>: ['__dict__',
                        '__doc__',
                        '__init__',
                        '__module__',
                        '__weakref__',
                        'z']}
```

Но нам необходимо двигаться дальше, чтобы лучше понять роль слотов — и осознать важность проверки в `mapattrs` на предмет наличия словаря `__dict__` перед его извлечением!

Изучите код, чтобы глубже вникнуть в его суть. Для инструмента вывода деревьев следующим шагом может быть индексирование словаря результатов функции `mapattrs` по `bysource=True` с целью получения атрибутов объекта при обходе древовидной структуры вместо текущего просмотра физического словаря `__dict__` (или возможно в дополнение к нему). Вероятно, для извлечения значений атрибутов придется использовать `getattr` на экземпляре, потому что некоторые из них могут быть реализованы как слоты или другие "виртуальные" атрибуты на уровне своих исходных классов, а их прямое извлечение на уровне класса не возвратит значение экземпляра. Тем не менее, если я начну приводить здесь еще больше кода, то лишу читателей предстоящего веселья и смысла читать следующий раздел.

Расширения в классах нового стиля

Помимо изменений, описанных в предыдущем разделе (откровенно говоря, некоторые из них могут показаться слишком академичными и неясными, чтобы иметь значение для многих читателей книги), классы нового стиля предоставляют несколько более развитых инструментов, имеющих более прямое и практическое применение — *слоты, свойства, дескрипторы* и т.д. В последующих разделах предлагается обзор добавочных средств, доступных для классов нового стиля в Python 2.X и всех классов в Python 3.X. В эту категорию расширений также входит атрибут `__mro__` и вызов

super, которые раскрывались в других местах — первый рассматривался в предыдущем разделе при исследовании изменения, а второй будет обсуждаться в конце главы, чтобы послужить в качестве более крупного учебного примера.

Слоты: объявления атрибутов

Присваивая последовательность строковых имен атрибутов специальному атрибуту `__slots__` класса, мы можем позволить классу нового стиля ограничивать набор допустимых атрибутов, которые будут иметь экземпляры этого класса, а также оптимизировать потребление памяти и возможно скорость работы программы. Однако, как мы увидим, слоты должны использоваться только в приложениях, в которых добавочная сложность очевидно оправдана. Они усложнят ваш код, способны усложнить или нарушить работу кода, с которым вы можете иметь дело, и требовать эффективного универсального ввода в действие.

Основы слотов

Для применения слотов присвойте последовательность строковых имен специальной переменной и атрибуту `__slots__` на верхнем уровне оператора `class`: только именам в списке `__slots__` можно присваивать значения как атрибутам экземпляров. Тем не менее, подобно всем именам в Python именам атрибутов экземпляров по-прежнему должны присваиваться значения, прежде чем на них можно будет ссылаться, несмотря на то, что они перечислены в `__slots__`:

```
>>> class limiter(object):
        __slots__ = ['age', 'name', 'job']

>>> x = limiter()
>>> x.age             # Перед использованием потребуется присвоить значение
AttributeError: age
Ошибка атрибута: age

>>> x.age = 40        # Выглядит подобно данным экземпляра
>>> x.age
40
>>> x.ape = 1000      # Недопустимое имя: не в __slots__
AttributeError: 'limiter' object has no attribute 'ape'
Ошибка атрибута: объект limiter не имеет атрибута ape
```

Средство слотов задумывалось как способ отлавливания опечаток вроде показанного выше (обнаруживается присваивание недопустимым именам атрибутов, которые отсутствуют в `__slots__`), а также как механизм оптимизации.

Размещение словаря пространств имен для каждого объекта экземпляра может оказаться дорогостоящим в плане памяти, если создается много экземпляров, а обязательных атрибутов лишь несколько. Чтобы сберечь пространство, Python не размещает словарь в каждом экземпляре, а резервирует для каждого *экземпляра* лишь пространство под хранение значения каждого атрибута слота и унаследованных атрибутов из общего *класса* для управления доступом к слотам. Такой прием способен дополнительно ускорить выполнение, хотя данное преимущество менее очевидно и может варьироваться в зависимости от программы, платформы и версии Python.

Слоты также являются своего рода крупным расхождением с основной динамической природой Python, которая требует, чтобы любое имя можно было создавать присваиванием. Фактически слоты имитируют язык C++ для эффективности за счет гибкости и даже обладают потенциалом *нарушать работу* ряда программ. Как будет показано, слоты также сопровождаются обилием правил для особых случаев исполь-

зования. Согласно собственной документации по Python они *не* должны применяться нигде, кроме как в очевидно гарантированных случаях – слоты трудно использовать корректно и, как указано в руководстве, их:

> лучше всего оставлять для тех редких случаев, когда имеется большое количество экземпляров в приложении, критичном к потреблению памяти.

Другими словами, это еще одно средство, которое должно применяться только при наличии четкой гарантии. К сожалению, слоты встречаются в коде на Python гораздо чаще, чем должны; их неясность сама по себе является недостатком. Как обычно, знание – ваш лучший союзник в подобных вещах, так что давайте кратко рассмотрим их.

Начиная с версии Python 3.3, требования к пространству атрибутов, *не хранящихся в слотах*, были понижены с помощью модели *словарей с разделяемыми ключами*, где словари `__dict__`, используемые для атрибутов объектов, могут разделять часть их внутреннего хранилища, в том числе хранилища ключей. Это может уменьшить ценность `__slots__` как инструмента оптимизации; согласно эталонным тестам такое изменение сокращает потребление памяти на 10–20% для объектно-ориентированных программ, показывает небольшое улучшение в скорости для программ, которые создают много похожих объектов, и в будущем вероятно продолжит оптимизироваться. С другой стороны, данное средство отнюдь не отменяет присутствия `__slots__` в существующем коде, с которым вам, возможно, предстоит разбираться!

Слоты и словари пространств имен

Оставив в стороне потенциальные преимущества, слоты способны значительно усложнить модель классов и опирающийся на нее код. На самом деле одни экземпляры со слотами могут вообще не содержать словаря пространств имен атрибутов `__dict__`, а другие будут иметь атрибуты данных, которые такой словарь не включает. Для ясности: это крупная *несовместимость* с традиционной моделью классов – такая, которая может усложнить любой код, получающий доступ к атрибутам обобщенным образом, и даже привести к полному отказу ряда программ.

Например, если есть вероятность применения слотов, тогда программам, которые составляют списки или обращаются к атрибутам экземпляра по строковым именам, может потребоваться использовать более нейтральные к хранилищу интерфейсы, нежели `__dict__`. Поскольку данные экземпляра могут включать имена уровня классов, такие как слоты – либо в дополнение, либо вместо хранилища словаря пространств имен, – для полноты может возникнуть необходимость опрашивать оба источника атрибутов.

Давайте посмотрим, что сказанное означает в переводе на код, и попутно получим больше сведений о слотах. Прежде всего, когда применяются слоты, экземпляры обычно не имеют словаря атрибутов – взамен интерпретатор Python использует представляемое позже средство *дескрипторов* класса для выделения и управления пространством, зарезервированным под слотовые атрибуты в экземпляре. Вот как это делается в Python 3.X и в Python 2.X для классов нового стиля, производных от `object`:

```
>>> class C:                        # В Python 2.X требуется (object)
        __slots__ = ['a', 'b']      # По умолчанию __slots__ означает
                                    #   отсутствие __dict__
>>> X = C()
>>> X.a = 1
```

```
>>> X.a
1
>>> X.__dict__
AttributeError: 'C' object has no attribute '__dict__'
Ошибка атрибута: объект C не имеет атрибута __dict__
```

Однако мы все еще можем извлекать и устанавливать атрибуты, основанные на слотах, по строковому имени с применением нейтральных к хранилищу инструментов, таких как getattr, setattr (просматривают за пределами __dict__ экземпляра и потому охватывают имена уровня класса вроде слотов) и dir (собирает все унаследованные имена повсюду в дереве):

```
>>> getattr(X, 'a')
1
>>> setattr(X, 'b', 2)    # Но getattr() и setattr() по-прежнему работают
>>> X.b
2
>>> 'a' in dir(X)         # И dir() тоже находит слотовые атрибуты
True
>>> 'b' in dir(X)
True
```

Также имейте в виду, что без словаря пространств имен атрибутов выполнять присваивание в экземплярах новым именам, отсутствующим в списке __slots__, невозможно:

```
>>> class D:                  # В Python 2.X использовать D(object)
        __slots__ = ['a', 'b']
        def __init__(self):
            self.d = 4        # В отсутствие __dict__ добавлять новые имена невозможно

>>> X = D()
AttributeError: 'D' object has no attribute 'd'
Ошибка атрибута: объект D не имеет атрибута d
```

Тем не менее, мы по-прежнему может разместить добавочные атрибуты, явно включая __dict__ в __slots__, чтобы создать также и словарь пространств имен атрибутов:

```
>>> class D:
        __slots__ = ['a', 'b', '__dict__']  # Указание __dict__ для его включения
        c = 3                               # Атрибуты класса работают нормально
        def __init__(self):
            self.d = 4                      # d хранится в __dict__, a является слотом

>>> X = D()
>>> X.d
4
>>> X.c
3
>>> X.a                                     # До присваивания все атрибуты экземпляра
                                            #   не определены
AttributeError: a
Ошибка атрибута: a
>>> X.a = 1
>>> X.b = 2
```

В рассматриваемом случае задействованы *оба* механизма хранения. Это делает словарь __dict__ слишком ограниченным для кода, в котором желательно трактовать

слоты как данные экземпляра, но обобщенные инструменты наподобие getattr все еще позволяют нам обрабатывать обе формы хранения как единый набор атрибутов:

```
>>> X.__dict__     # Некоторые объекты имеют и имена __dict__, и слотовые имена
{'d': 4}           # getattr() может извлекать любой из двух типов атрибутов
>>> X.__slots__
['a', 'b', '__dict__']
>>> getattr(X, 'a'), getattr(X, 'c'), getattr(X, 'd')  # Извлекает все 3 формы
(1, 3, 4)
```

Однако поскольку инструмент dir также возвращает все *унаследованные* атрибуты, в ряде контекстов он может оказаться чересчур обширным; вдобавок он включает методы уровня класса и даже все стандартные методы object. В коде, где нужно перечислять *только* атрибуты экземпляра, по-прежнему необходимо явно разрешить обе формы хранения. Мы могли бы сначала написать наивный код вроде приведенного ниже:

```
>>> for attr in list(X.__dict__) + X.__slots__:    # Неправильно...
        print(attr, '=>', getattr(X, attr))
```

Так как любая из двух форм хранения может быть опущена, мы могли бы написать более корректный код, используя getattr, чтобы разрешить стандартные методы — замечательный, но все-таки неточный подход, как объясняется в следующем разделе:

```
>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []):
        print(attr, '=>', getattr(X, attr))
d => 4
a => 1    # Менее неправильно...
b => 2
__dict__ => {'d': 4}
```

Множество списков __slots__ в суперклассах

Предыдущий код работает в этом особом случае, но в целом он *не полностью точен*. В частности, код обращается к слотовым именам только в *самом нижнем* атрибуте __slots__, который унаследован экземпляром, но списки __slots__ могут появляться в дереве классов более одного раза. То есть отсутствие имени в само нижнем списке __slots__ вовсе не препятствует его существованию в более высоком списке __slots__. Из-за того, что имена становятся атрибутами уровня класса, экземпляры обзаводятся объединением всех слотовых имен везде в дереве по нормальному правилу наследования:

```
>>> class E:
        __slots__ = ['c', 'd']            # Суперкласс имеет слоты
>>> class D(E):
        __slots__ = ['a', '__dict__']     # Но его подкласс тоже

>>> X = D()
>>> X.a = 1; X.b = 2; X.c = 3   # Экземпляр получает объединение (слоты: a, c)
>>> X.a, X.c
(1, 3)
```

Инспектирование только списка унаследованных слотов не приведет к выбору слотов, определенных выше в дереве классов:

```
>>> E.__slots__          # Однако слоты не объединяются
['c', 'd']
>>> D.__slots__
['a', '__dict__']
```

```
>>> X.__slots__         # Экземпляр наследует *самый нижний* список __slots__
['a', '__dict__']
>>> X.__dict__          # И имеет собственный словарь атрибутов
{'b': 2}
>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []):
        print(attr, '=>', getattr(X, attr))
b => 2                  # Слоты остальных суперклассов отсутствуют!
a => 1
__dict__ => {'b': 2}
>>> dir(X)              # Но dir() включает все слотовые имена
[...много имен не показано... 'a', 'b', 'c', 'd']
```

Другими словами, что касается обобщенного перечисления, то атрибутов одного списка `__slots__` не всегда достаточно – потенциально они подвергаются полной процедуре поиска при наследовании. Еще один пример слотов, появляющихся в нескольких суперклассах, демонстрировался в файле `mapattrs-slots.py` ранее в главе. Если множество классов в дереве имеют собственные атрибуты `__slots__`, тогда обобщенные программы должны внедрять другие политики для перечисления атрибутов, что объясняется в следующем разделе.

Обобщенная обработка слотов и других "виртуальных" атрибутов

Вероятно, в этот момент у вас возникнет желание снова просмотреть варианты реализации политики слотов, которые приводились ближе к концу предыдущей главы при обсуждении подмешиваемых классов отображения из файла `lister.py` – основной пример того, почему обобщенные программы должны позаботиться о слотах. Инструменты, пытающиеся обобщенным образом строить списки атрибутов данных экземпляров, обязаны учитывать слоты и возможно другие "виртуальные" атрибуты экземпляров, подобные *свойствам* и *дескрипторам* – имена, которые тоже располагаются в классах, но могут предоставлять значения атрибутов для экземпляров по запросу. Слоты ориентированы на данные, но являются типичным представителем этой более широкой категории.

Такие атрибуты требуют инклюзивных подходов, специальной обработки или общего игнорирования – последний вариант становится неприемлемым, как только любой программист начнет применять слоты в своем прикладном коде. По правде говоря, атрибуты экземпляров уровня классов, такие как слоты, неизбежно влекут за собой переопределение понятия *данных экземпляра* – как локально хранящихся атрибутов, объединения всех унаследованных атрибутов или какого-то их подмножества.

Например, некоторые программы могут относить слотовые имена к атрибутам *классов*, а не экземпляров; в конце концов, эти атрибуты не находятся в словарях пространств имен экземпляров. В качестве альтернативы, как было показано ранее, программы могут быть более инклюзивными за счет извлечения с помощью `dir` всех имен унаследованных атрибутов и получения посредством `getattr` соответствующих атрибутам значений для экземпляра – независимо от их физического местоположения или реализации. Если вы должны поддерживать слоты как данные экземпляра, тогда продемонстрированный далее способ, скорее всего, окажется самым надежным:

```
>>> class Slotful:
        __slots__ = ['a', 'b', '__dict__']
        def __init__(self, data):
            self.c = data

>>> I = Slotful(3)
```

```
>>> I.a, I.b = 1, 2
>>> I.a, I.b, I.c           # Нормальное извлечение атрибутов
(1, 2, 3)
>>> I.__dict__              # Присутствуют хранилища __dict__ и __slots__
{'c': 3}
>>> [x for x in dir(I) if not x.startswith('__')]
['a', 'b', 'c']
>>> I.__dict__['c']         # Единственным источником атрибутов является __dict__
3
>>> getattr(I, 'c'), getattr(I, 'a')    # Сочетание dir+getattr обширнее,
                                        #   чем просто __dict__
(3, 1)                      # Применяется к слотам, свойствам, дескрипторам
>>> for a in (x for x in dir(I) if not x.startswith('__')):
        print(a, getattr(I, a))

a 1
b 2
c 3
```

В рамках такой модели dir/getattr вы по-прежнему можете отображать атрибуты на их источники наследования и при необходимости фильтровать их более избирательно по источнику или типу, просматривая *MRO* – как мы поступали ранее с инструментами в mapattrs.py и их приложением к слотам в mapattrs-slots.py. В качестве добавочного бонуса инструменты подобного рода и политики для обработки слотов потенциально будут автоматически применяться также к *свойствам* и *дескрипторам*, хотя по сравнению со слотами упомянутые атрибуты больше являются вычисляемыми значениями и в меньшей степени – данными, связанными с экземплярами.

Также имейте в виду, что это не просто проблема с инструментами. Атрибуты экземпляров, основанные на классах, вроде слотов также оказывают влияние на традиционное кодирование метода перегрузки операции __setattr__, который мы встречали в главе 30. Поскольку слоты и ряд других атрибутов не хранятся в словаре __dict__ экземпляра и могут даже стать причиной его *отсутствия*, классы нового стиля взамен обычно должны выполнять присваивания атрибутов, направляя их суперклассу object. По указанной причине на практике данный метод может фундаментально отличаться в некоторых классических классах и классах нового стиля.

Правила использования слотов

Объявления слотов могут обнаруживаться во многих классах в дереве классов. Тем не менее, в таком случае с ними связаны ограничения, которым довольно сложно давать рациональное объяснение, если только вы не воспримете реализацию слотов как *дескрипторов* уровня класса для каждого слотового имени, унаследованного экземплярами, где резервируется управляемое пространство (дескрипторы подробно рассматриваются в последней части книги).

- **Слоты в подклассах бессмысленны, когда они отсутствуют в суперклассах.** Если подкласс унаследован от суперкласса без __slots__, то атрибут __dict__ экземпляра, созданный для суперкласса, будет всегда доступен, делая атрибут __slots__ в подклассе по существу бессмысленным. Подкласс по-прежнему управляет своими слотами, но никак не вычисляет их значения и не избегает словаря – главной причины применения слотов.

- **Слоты в суперклассах бессмысленны, когда они отсутствуют в подклассах.** Аналогично, поскольку объявление `__slots__` ограничено классом, в котором оно появляется, подклассы будут создавать `__dict__` экземпляра, если в них не определен атрибут `__slots__`, делая `__slots__` в суперклассе в сущности бессмысленным.
- **Переопределение делает бессмысленными слоты суперкласса.** Если класс определяет такое же слотовое имя, как в суперклассе, то согласно нормальному наследованию его переопределение скрывает слот из суперкласса. Вы можете получить доступ к версии имени, которая определена в слоте суперкласса, только путем извлечения его дескриптора напрямую из суперкласса.
- **Слоты препятствуют определению стандартных имен.** Поскольку слоты реализованы в виде дескрипторов уровня класса (вместе с пространством для каждого экземпляра), вы не можете использовать атрибуты класса с такими же именами для предоставления стандартных имен, как могли бы делать для нормальных атрибутов экземпляра: присваивание значения тому же самому имени в классе переопределяет дескриптор слота.
- **Слоты и `__dict__`.** Как было показано ранее, список `__slots__` предотвращает существование `__dict__` экземпляра и присваивание значений именам, отсутствующим в списке, если `__dict__` явно не включен в список.

Последний пункт мы уже видели в действии, а третий пункт иллюстрировался ранее в файле `mapattrs-slots.py`. Несложно продемонстрировать, каким образом новые правила воплощаются в коде — наиболее критично то, что словарь пространств имен создается, когда в любом классе в дереве опущены слоты, тем самым сводя на нет выигрыш от оптимизации памяти:

```
>>> class C: pass            # Пункт 1: слоты в подклассе, но не в суперклассе
>>> class D(C): __slots__ = ['a'] # Создает словарь экземпляра для неслотовых имен
>>> X = D()                 # Но слотовое имя по-прежнему поддерживается в классе
>>> X.a = 1; X.b = 2
>>> X.__dict__
{'b': 2}
>>> D.__dict__.keys()
dict_keys([... 'a', '__slots__', ...])

>>> class C: __slots__ = ['a']  # Пункт 2: слоты в суперклассе, но не в подклассе
>>> class D(C): pass             # Создает словарь экземпляра для неслотовых имен
>>> X = D()                 # Но слотовое имя по-прежнему поддерживается в классе
>>> X.a = 1; X.b = 2
>>> X.__dict__
{'b': 2}
>>> C.__dict__.keys()
dict_keys([... 'a', '__slots__', ...])

>>> class C: __slots__ = ['a'] # Пункт 3: доступен только самый нижний слот
>>> class D(C): __slots__ = ['a']

>>> class C: __slots__ = ['a']; a = 99   # Пункт 4: стандартные имена уровня
                                        #  класса отсутствуют
ValueError: 'a' in __slots__ conflicts with class variable
Ошибка значения: a в __slots__ конфликтует с переменной класса
```

Другими словами, кроме потенциала нарушить работу программ слоты по существу требуют *как универсального, так и тщательного ввода в действие*, чтобы быть эффективными — поскольку слоты не вычисляют значения динамически подобно свойствам

(рассматриваются в следующем разделе), по большому счету они нецелесообразны, если не применяются каждым классом в дереве и осмотрительно определяют только новые слотовые имена, не определяемые другими классами. Это средство используется по принципу "все или ничего" – неудачное свойство, разделяемое обсуждаемым позже вызовом super:

```
>>> class C: __slots__ = ['a']       # Предполагает универсально используемые,
                                      #   отличающиеся имена
>>> class D(C): __slots__ = ['b']
>>> X = D()
>>> X.a = 1; X.b = 2
>>> X.__dict__
AttributeError: 'D' object has no attribute '__dict__'
Ошибка атрибута: объект D не имеет атрибута __dict__
>>> C.__dict__.keys(), D.__dict__.keys()
(dict_keys([... 'a', '__slots__', ...]), dict_keys([... 'b', '__slots__', ...]))
```

Такие правила (среди прочих, касающихся *слабых ссылок*, которые ради экономии места здесь не рассматриваются) являются частью причины, почему применять слоты обычно не рекомендуется кроме патологических случаев, где обеспечиваемое ими уменьшение пространства памяти считается важным. Да и то, их предрасположенность к усложнению кода и нарушению его работы должна быть достаточным основанием для того, чтобы внимательно относиться к компромиссам. Слоты не только должны повсеместно распространяться по структуре, но они могут также привести к отказу инструментов, на которые вы полагаетесь.

Примеры влияния слотов: ListTree и mapattrs

В качестве более реалистичного примера воздействия слотов отметим, что из-за первого пункта списка в предыдущем разделе класс ListTree, реализованный в главе 31, *не терпит отказ*, когда подмешивается к классу, который определяет список __slots__, хотя он просматривает словари пространств имен экземпляров. Отсутствия слотов в самом классе ListTree достаточно для того, чтобы гарантировать, что экземпляр по-прежнему будет иметь словарь __dict__ и, следовательно, исключение не сгенерируется при извлечении либо индексировании. Скажем, оба показанных ниже класса выполняют отображение без ошибок – второй фрагмент также разрешает присваивать значения именам не из списка __slots__ как атрибутам экземпляров, в том числе любым именам, которые требует суперкласс:

```
class C(ListTree): pass
X = C()                          # Нормально: списки __slots__ не используются
print(X)

class C(ListTree): __slots__ = ['a', 'b'] # Нормально: суперкласс создает __dict__
X = C()
X.c = 3
print(X)
```

Следующие классы также выполняют отображение корректно -- *любой* класс без __slots__ вроде ListTree создает __dict__ экземпляра и потому может безопасно предположить его наличие:

```
class A: __slots__ = ['a']      # Оба класса работают нормально согласно пункту 1
class B(A, ListTree): pass

class A: __slots__ = ['a']
class B(A, ListTree): __slots__ = ['b']     # Отображает b в B, a в A
```

Несмотря на то что это делает слоты подклассов бесполезными, оно будет положительным побочным эффектом для таких классов инструментов, как ListTree (и его предшественник из главы 28). Однако в общем случае некоторые инструменты могут требовать перехвата исключений, когда __dict__ отсутствует, либо использовать hasattr или getattr для проверки или предоставления стандартных имен, если применение слотов способно препятствовать инспектированию словаря пространств имен в объектах экземпляров.

Например, теперь вы должны понимать, почему в программе mapattrs.py, представленной ранее в главе, требовалась проверка на предмет присутствия словаря __dict__ перед его извлечением — объекты экземпляров, созданные из классов со списками __slots__, не имеют __dict__. На самом деле, если мы будем использовать выделенную альтернативную строку, как показано далее, то функция mapattrs потерпит неудачу с генерацией исключения при попытке поиска имени атрибута в экземпляре в начале последовательности пути наследования:

```
def mapattrs(instance, withobject=False, bysource=False):
    for attr in dir(instance):
        for obj in inherits:
            if attr in obj.__dict__:    # Может потерпеть неудачу,
                                        #   если используется __slots__

>>> class C: __slots__ = ['a']
>>> X = C()
>>> mapattrs(X)
AttributeError: 'C' object has no attribute '__dict__'
Ошибка атрибута: объект C не имеет атрибута __dict__
```

Любая из следующих двух строк кода обходит проблему и позволяет инструменту поддерживать слоты — первая предоставляет стандартное имя, а вторая длиннее, но выглядит чуть более явной в своем намерении:

```
if attr in getattr(obj, '__dict__', {}):

if hasattr(obj, '__dict__') and attr in obj.__dict__:
```

Как упоминалось ранее, некоторые инструменты могут извлекать преимущество от подобного отображения результатов dir на объекты в MRO вместо просмотра __dict__ экземпляра в целом — без такого более инклюзивного подхода атрибуты, реализованные инструментами уровня классов вроде слотов, не будут относиться к данным экземпляра. Но даже в указанных обстоятельствах это не обязательно освобождает подобные инструменты от необходимости учитывать отсутствующий словарь __dict__ в экземпляре!

Как насчет скорости работы слотов?

Наконец, в то время как слоты главным образом оптимизируют потребление памяти, их влияние на скорость работы менее ярко выражено. Ниже приведен простой тестовый сценарий, в котором применяются методики timeit, предложенные в главе 21. Для обеих моделей хранения со слотами и без слотов (словарь экземпляра) он создает 1000 экземпляров, для каждого экземпляра присваивает значение 4 атрибутам, после чего извлекает их и повторяет процесс 1000 раз; затем для обеих моделей отбираются по 3 лучших результата выполнения 8 миллионов операций с атрибутами:

```
# Файл slots-test.py
from __future__ import print_function
import timeit
```

```
base = """
Is = []
for i in range(1000):
   X = C()
   X.a = 1; X.b = 2; X.c = 3; X.d = 4
   t = X.a + X.b + X.c + X.d
   Is.append(X)
"""

stmt = """
class C(object):
   __slots__ = ['a', 'b', 'c', 'd']
""" + base
print('Slots =>', end=' ')      # Со слотами
print(min(timeit.repeat(stmt, number=1000, repeat=3)))

stmt = """
class C(object):
   pass
""" + base
print('Nonslots=>', end=' ')    # Без слотов
print(min(timeit.repeat(stmt, number=1000, repeat=3)))
```

Во всяком случае, с таким кодом и установленными версиями Python 3.7 и Python 2.7 по лучшим показателям времени можно сделать вывод, что слоты слегка быстрее в обеих линейках Python 3.X и Python 2.X. Правда, результаты мало говорят о расходе памяти и подвержены произвольным изменениям в будущем:

```
c:\code> py -3 slots-test.py
Slots   => 0.9774146349999999
Nonslots=> 1.1963583829999997

C:\code> py -2 slots-test.py
Slots   => 0.615521153591
Nonslots=> 0.766582559582
```

Дополнительные сведения о слотах в целом ищите в стандартном наборе руководств по Python. Также дождитесь учебного примера с декоратором `Private` в главе 39, который естественным образом допускает атрибуты, основанные на хранилищах `__slots__` и `__dict__`, за счет использования делегирования и инструментов доступа, нейтральных к хранилищу, наподобие `getattr`.

Свойства: средства доступа к атрибутам

Нашим следующим расширением нового стиля являются *свойства* — механизм, который снабжает классы нового стиля еще одним способом определения методов, автоматически вызываемых для доступа к атрибутам экземпляра или присваивания им значений. Это средство похоже на свойства (они же методы получения и установки) в языках вроде Java и C#, но в Python его лучше применять обычно умеренно в качестве способа для добавления средств доступа к атрибутам *после* того, как потребности развиваются и оправдываются. Тем не менее, при необходимости свойства позволяют динамически вычислять значения атрибутов, не требуя вызова методов в точке доступа.

Хотя свойства не могут поддерживать обобщенные цели маршрутизации атрибутов, по крайней мере, для специфических атрибутов они являются альтернативой традиционным вариантам использования методов для перегрузки операций `__getattr__` и `__setattr__`, с которыми мы впервые сталкивались в главе 30. Свойства обладают эффектом, который аналогичен упомянутым двум методам, но по контрасту предус-

матривают добавочный вызов метода только для доступа к именам, требующим динамического вычисления – доступ к другим, отличным от свойств именам обычно происходит без дополнительных вызовов. В то время как метод `__getattr__` вызывается только для *неопределенных* имен, метод `__setattr__` вызывается для присваивания значения *каждому* атрибуту.

Свойства и слоты тоже связаны друг с другом, но служат разным целям. Они оба реализуют атрибуты экземпляров, которые физически не хранятся в словарях пространств имен экземпляров – своего рода "виртуальные" атрибуты – и основаны на понятии *дескрипторов* атрибутов уровня класса. Однако слоты управляют хранилищем экземпляра, тогда как свойства перехватывают операции доступа и вычисляют значения произвольным образом. Поскольку внутренняя реализация слишком сложна, чтобы раскрывать ее здесь, свойства и дескрипторы будут рассматриваться в главе 38.

Основы свойств

В качестве краткого введения отметим, что свойство представляет собой тип объекта, присвоенного имени атрибута класса. Для создания свойства необходимо вызвать встроенную функцию `property` и передать ей три метода доступа (обработчики операций получения, установки и удаления), а также необязательную строку документации для свойства. Если любой из аргументов опущен или для него передается `None`, то связанная с ним операция не поддерживается.

Результирующий объект свойства обычно присваивается имени на верхнем уровне оператора `class` (например, *имя*=`property()`), и для автоматизации этого шага доступен специальный синтаксис `@`, с которым мы встретимся позже. При таком присваивании последующие обращения к имени свойства класса как к атрибуту объекта (скажем, *объект.имя*) автоматически направляются одному из методов доступа, которые передавались вызову `property`.

Например, ранее было показано, что метод перегрузки операции `__getattr__` позволяет классам перехватывать ссылки на неопределенные атрибуты в классических классах и в классах нового стиля:

```
>>> class operators:
        def __getattr__(self, name):
            if name == 'age':
                return 40
            else:
                raise AttributeError(name)

>>> x = operators()
>>> x.age                  # Запускается __getattr__
40
>>> x.name                 # Запускается __getattr__
AttributeError: name
Ошибка атрибута: name
```

Ниже приведен тот же пример, реализованный с применением свойств. Обратите внимание, что свойства доступны во всех классах, но в Python 2.X требуют наследования нового стиля от `object`, чтобы надлежащим образом работать для перехвата операций *присваивания* значений атрибутам (если вы забудете об этом, то свойство без каких-либо уведомлений переустановится в новые данные):

```
>>> class properties(object):                      # Для методов установки в
Python 2.X нужен object
        def getage(self):
```

```
        return 40
    age = property(getage, None, None, None)    # (получение, установка,
                        #  удаление, строка документации) либо использовать @
>>> x = properties()
>>> x.age               # Запускается getage
40
>>> x.name              # Нормальное извлечение
AttributeError: 'properties' object has no attribute 'name'
Ошибка атрибута: объект properties не имеет атрибута name
```

Для ряда кодовых задач свойства могут оказаться менее сложными и более быстрыми при выполнении, чем традиционные методики. Скажем, когда мы добавляем поддержку *присваивания* значений атрибутам, то свойства становятся более привлекательными – они требуют меньшего объема кода, а для операций присваивания значений атрибутам, которые нежелательно вычислять динамически, не возникают избыточные вызовы методов:

```
>>> class properties(object): # Для методов установки в Python 2.X нужен object
    def getage(self):
        return 40
    def setage(self, value):
        print('set age: %s' % value)
        self._age = value
    age = property(getage, setage, None, None)
>>> x = properties()
>>> x.age               # Запускается getage
40
>>> x.age = 42          # Запускается setage
set age: 42
>>> x._age              # Нормальное извлечение: getage не вызывается
42
>>> x.age               # Запускается getage
40
>>> x.job = 'trainer' # Нормальное присваивание: setage не вызывается
>>> x.job               # Нормальное извлечение: getage не вызывается
'trainer'
```

Эквивалентный класс, основанный на перегрузке операций, влечет за собой добавочные вызовы методов для присваивания значений атрибутам, которыми он не управляет. Кроме того, во избежание зацикливания он нуждается в перенаправлении таких операций присваивания словарю атрибутов (или методу `__setattr__` суперкласса `object` в случае классов нового стиля для лучшей поддержки "виртуальных" атрибутов вроде слотов и свойств, реализованных в других классах):

```
>>> class operators:
    def __getattr__(self, name):    # При ссылке на неопределенный атрибут
        if name == 'age':
            return 40
        else:
            raise AttributeError(name)
    def __setattr__(self, name, value):  # При всех операциях присваивания
        print('set: %s %s' % (name, value))
        if name == 'age':
            self.__dict__['_age'] = value  # Или object.__setattr__()
        else:
            self.__dict__[name] = value
```

```
>>> x = operators()
>>> x.age                           # Запускается __getattr__
40
>>> x.age = 41                      # Запускается __setattr__
set: age 41
>>> x._age                          # Определен: __getattr__ не вызывается
41
>>> x.age                           # Запускается __getattr__
40
>>> x.job = 'trainer'               # Снова запускается __setattr__
set: job trainer
>>> x.job                           # Определен: __getattr__ не вызывается
'trainer'
```

Свойства выглядят как выигрыш в приведенном простом примере. Тем не менее, некоторые приложения `__getattr__` и `__setattr__` по-прежнему требуют более динамичных или обобщенных интерфейсов, чем те, которые свойства обеспечивают напрямую.

Например, во многих случаях набор поддерживаемых атрибутов не удается определить при реализации класса, и он даже может не существовать в какой-либо материальной форме (например, когда ссылки на произвольные атрибуты *делегируются* внутреннему/вложенному объекту обобщенным образом). В таких контекстах обычно предпочтительнее иметь обобщенный обработчик атрибутов `__getattr__` или `__setattr__` с передаваемым именем атрибута. Поскольку обобщенные обработчики подобного рода способны также поддерживать более простые сценарии, свойства нередко оказываются необязательным и избыточным расширением — хотя оно позволяет избежать избыточных вызовов при присваивании и некоторые программисты могут отдавать ему предпочтение, когда оно применимо.

Более подробные сведения об обоих вариантах будут предложены в главе 38 последней части книги. Как там будет показано, свойства также возможно реализовывать с использованием *синтаксиса декораторов функций* в виде символа `@`, который описан позже в главе; он является эквивалентом и автоматической альтернативой ручному присваиванию на уровне объявления класса:

```
class properties(object):
    @property                       # Реализация свойств с помощью
                                    #   декораторов: обсуждается позже
    def age(self):
        ...
    @age.setter
    def age(self, value):
        ...
```

Чтобы понять показанный синтаксис декораторов, нам нужно двигаться дальше.

Метод `__getattribute__` и дескрипторы: инструменты для работы с атрибутами

Также входя в состав расширений классов, метод перегрузки операции `__getattribute__`, доступный только для классов нового стиля, позволяет классу перехватывать ссылки на *все* атрибуты, а не только на неопределенные. Это делает его мощнее родственного метода `__getattr__`, который мы применяли в предыдущем разделе, но также и более сложным в использовании — во многом подобно `__setattr__` он предрасположен к зацикливанию, но по-другому.

Для более специализированных целей перехвата доступа к атрибутам вдобавок к свойствам и методам перегрузки операций Python поддерживает понятие *дескрипторов* атрибутов – классов с методами `__get__` и `__set__`, которые присваиваются атрибутам класса, наследуются экземплярами и предназначены для перехвата доступа по чтению и записи к специфическим атрибутам. Ради демонстрации ниже представлен один из простейших дескрипторов, какой только можно встретить:

```
>>> class AgeDesc(object):
        def __get__(self, instance, owner): return 40
        def __set__(self, instance, value): instance._age = value

>>> class descriptors(object):
        age = AgeDesc()

>>> x = descriptors()
>>> x.age                # Запускается AgeDesc.__get__
40
>>> x.age = 42           # Запускается AgeDesc.__set__
>>> x._age               # Нормальное извлечение: AgeDesc не вызывается
42
```

Дескрипторы имеют доступ к состоянию в экземплярах самих себя, а также своего клиентского класса, и в некотором смысле являются более универсальной формой свойств; на самом деле свойства представляют собой упрощенный способ определения специфического типа дескриптора – того, что запускает функции при доступе. Кроме того, дескрипторы применяются для реализации описанного ранее средства слотов и других инструментов Python.

Поскольку тема, связанная с методом `__getattribute__` и дескрипторами, слишком обширна, чтобы как следует раскрыть ее здесь, мы отложим ее до главы 38, где приведем дополнительные сведения о свойствах. Мы также задействуем эти средства в примерах в главе 39 и выясним, как они учитывают наследование, в главе 40.

Другие изменения и расширения классов

Как уже упоминалось, мы также откладываем до конца главы исследование встроенной функции `super` – дополнительного крупного расширения классов нового стиля, которое полагается на MRO. Однако прежде чем достичь своей цели, нам необходимо ознакомиться с другими изменениями и расширениями, касающимися классов, которые не обязательно связаны с классами нового стиля, но были введены приблизительно в то же самое время: статические методы и методы классов, декораторы и т.д.

Многие изменения и добавленные средства классов нового стиля интегрируются с понятием типов, допускающих создание подклассов, потому что такие типы и классы нового стиля появились в сочетании с объединением типов и классов в Python 2.2 и последующих версиях. Как было показано, в Python 3.X объединение завершилось: теперь классы являются типами, а типы – классами, и классы Python в наши дни все еще отражают как концептуальное объединение, так и его реализацию.

Наряду с указанными изменениями в Python также сформировался более согласованный и обобщенный протокол для создания *метаклассов* – классов, которые образуют подклассы типа `type`, перехватывают вызовы создания классов и способны предоставлять линии поведения, приобретаемые классами. Соответственно, они обеспечивают четко определенную привязку для управления и дополнения объектов классов. Метаклассы также представляют собой сложную тему, знание которой необязательно для большинства программистов на Python, поэтому мы не будем здесь затра-

гивать дополнительные детали. Позже в главе мы снова бегло взглянем на метаклассы в сочетании с декораторами классов — средством, роли которого часто перекрываются и которое полностью рассматривается в главе 40. А теперь давайте перейдем к нескольким расширениям, связанным с классами.

Статические методы и методы классов

Начиная с версии Python 2.2, появилась возможность определять внутри класса два вида методов, допускающих вызов без экземпляра: *статические* методы работают примерно так же, как простые не имеющие экземпляров функции в классе, а методам *класса* вместо экземпляра передается класс. Оба похожи на инструменты в других языках (скажем, на статические методы в C++). Хотя статические методы и методы классов были добавлены вместе с классами нового стиля, которые обсуждались в предшествующих разделах, они работают также и с классическими классами.

Чтобы включить данные режимы методов, понадобится вызвать специальные встроенные функции с именами `staticmethod` и `classmethod` внутри класса или обратиться к ним с помощью особого синтаксиса декораторов `@имя`, рассматриваемого позже в главе. Указанные функции обязательны для включения специальных режимов методов в Python 2.X и в целом необходимы в Python 3.X. В Python 3.X объявление `staticmethod` не является обязательным для методов уровня экземпляра, вызываемых только через имя класса, но все-таки требуется, если такие методы вызываются через экземпляры.

Для чего используются специальные методы?

Как уже известно, методу класса в первом аргументе обычно передается объект экземпляра, чтобы служить подразумеваемым объектом вызова метода — это "объект" в "объектно-ориентированном программировании". Тем не менее, в наши дни существуют два способа модификации такой модели. Прежде, чем выяснить, что они собой представляют, необходимо понять причины их важности.

Иногда программы должны обрабатывать данные, ассоциированные с классами, а не экземплярами. Подумайте об отслеживании количества экземпляров, созданных из класса, или ведении списка всех экземпляров класса, которые в текущий момент находятся в памяти. Информация подобного рода и ее обработка связаны с классом, но не с его экземплярами. То есть такая информация, как правило, хранится в самом классе и обрабатывается отдельно от любого экземпляра.

Для указанных задач часто может оказаться достаточно простых функций, реализованных за пределами класса — поскольку они могут обращаться к атрибутам класса через имя класса, то имеют доступ к данным класса и никогда не требуют доступа к какому-либо экземпляру. Однако чтобы лучше связать такой код с классом и сделать возможной обычную настройку обработки посредством наследования, было бы лучше реализовывать функции этого вида *внутри* самого класса. Таким образом, нам нужны в классе методы, которым не передается аргумент экземпляра `self` и они не ожидают его.

Цели подобного рода поддерживаются в Python с помощью понятия *статических методов* — простых функций без аргумента, которые вкладываются внутрь класса и рассчитаны на работу с атрибутами класса, а не экземпляра. Статические методы никогда не получают автоматический аргумент `self`, вызываются они через класс или через экземпляр, и по обыкновению отслеживают информацию, охватывающую все экземпляры, взамен предоставления линий поведения для экземпляров.

Несмотря на менее частое применение, в Python также поддерживается понятие *методов класса* – методов, которым в первом аргументе вместо объекта экземпляра передается объект класса, независимо от того, вызываются они через экземпляр или через класс. Такие методы могут получать доступ к данным класса посредством своего аргумента класса, который мы до сих пор называли `self`, даже если вызываются через экземпляр. Нормальные методы, теперь известные в формальных кругах как *методы экземпляров*, при вызове по-прежнему получают объект экземпляра, но статические методы и методы класса – нет.

Статические методы в Python 2.X и 3.X

Концепция статических методов одинакова в линейках Python 2.X и 3.X, но требования к их реализации в Python 3.X получили некоторое развитие. Поскольку в книге раскрываются обе версии, до того, как мы займемся написанием кода, необходимо рассмотреть отличия между двумя лежащими в основе моделями.

Вообще говоря, мы уже начали эту историю в предыдущей главе, когда исследовали понятие несвязанных методов. Вспомните, что в обеих линейках Python 2.X и 3.X методу, вызванному через экземпляр, всегда передается данный экземпляр. Тем не менее, методы, извлекаемые из класса напрямую, в Python 3.X трактуются иным образом, чем в Python 2.X – отличие между линейками Python, которое не имеет ничего общего с классами нового стиля.

- Обе линейки Python 2.X и 3.X производят связанные методы, когда метод извлекается через экземпляр.
- В Python 2.X извлечение метода из класса производит несвязанный метод, который не может быть вызван без передачи экземпляра вручную.
- В Python 3.X извлечение метода из класса производит простую функцию, которая может быть вызвана нормально в отсутствие какого-либо экземпляра.

Другими словами, методы классов в Python 2.X всегда требуют передачи экземпляра независимо от того, вызываются они через экземпляр или через класс. Напротив, в Python 3.X мы обязаны передавать экземпляр методу, только если он его ожидает – методы, не включающие аргумент экземпляра, могут вызываться через класс без передачи экземпляра. То есть Python 3.X разрешает присутствие в классе простых функций при условии, что они не ожидают аргумента экземпляра, и он им не передается. Ниже описан совокупный эффект.

- В Python 2.X мы должны всегда объявлять метод как статический, чтобы вызывать его без экземпляра через класс или через экземпляр.
- В Python 3.X нам не нужно объявлять такие методы как статические, если они будут вызываться только через класс, но мы обязаны делать это, чтобы вызывать их через экземпляр.

В целях иллюстрации предположим, что мы хотим использовать атрибуты класса для подсчета созданных экземпляров. Первая попытка предпринята в файле `spam_class.py`, содержимое которого показано ниже – его класс содержит счетчик, хранящийся в виде атрибута класса, конструктор, увеличивающий счетчик на единицу каждый раз, когда создается новый экземпляр, и метод, отображающий значение счетчика. Не забывайте, что атрибуты класса разделяются всеми экземплярами.

Следовательно, хранение счетчика в самом объекте класса гарантирует, что он охватывает все экземпляры:

```
class Spam:
   numInstances = 0
   def __init__(self):
      Spam.numInstances = Spam.numInstances + 1
   def printNumInstances():
      print("Number of instances created: %s" % Spam.numInstances)
            # Количество созданных экземпляров
```

Метод printNumInstances предназначен для обработки данных класса, не экземпляра – он касается *всех* экземпляров, а не какого-то конкретного, и потому мы хотим иметь возможность вызывать его без необходимости в передаче экземпляра. На самом деле для извлечения количества экземпляров создавать новый экземпляр нежелательно, ведь в результате счетчик экземпляров изменится! Таким образом, нам нужен "статический" метод без аргумента self.

Однако работает код printNumInstances или нет, зависит от применяемой версии Python и от способа вызова метода – через класс или через экземпляр. В Python 2.X вызовы функции метода без аргумента self через класс и через экземпляр потерпит неудачу (часть сообщений об ошибках не показана):

```
C:\code> c:\python27\python
>>> from spam_class import Spam
>>> a = Spam()    # Вызов несвязанных методов класса в Python 2.X невозможен
>>> b = Spam()    # По умолчанию методы ожидают объекта self
>>> c = Spam()

>>> Spam.printNumInstances()
TypeError: unbound method printNumInstances() must be called with
Spam instance as first argument (got nothing instead)
Ошибка типа: несвязанный метод printNumInstances() должен вызываться с
экземпляром Spam в первом аргументе (ничего не предоставлено взамен)
>>> a.printNumInstances()
TypeError: printNumInstances() takes no arguments (1 given)
Ошибка типа: printNumInstances() не принимает аргументов (предоставлен 1)
```

Проблема здесь в том, что несвязанные методы экземпляров – не в точности то же, что и простые функции в Python 2.X. Хотя в заголовке def не указано ни одного аргумента, при вызове метод все же ожидает передачи экземпляра, т.к. функция ассоциирована с классом. В Python 3.X вызовы методов без аргумента self через классы работают, но вызовы через экземпляры терпят неудачу:

```
C:\code> c:\python37\python
>>> from spam_class import Spam
>>> a = Spam()           # В Python 3.X можно вызывать функции в классе
>>> b = Spam()           # Вызовам через экземпляры по-прежнему передается self
>>> c = Spam()

>>> Spam.printNumInstances()       # В Python 3.X отличается
Number of instances created:s 3
>>> a.printNumInstances()
TypeError: printNumInstances() takes 0 positional arguments but 1 was given
Ошибка типа: printNumInstances() принимает 0 позиционных аргументов, но был
предоставлен 1
```

То есть вызовы не принимающих экземпляр методов вроде printNumInstances, сделанные через *класс*, отказывают в Python 2.X, но работают в Python 3.X.

С другой стороны, вызовы, произведенные через *экземпляр*, терпят неудачу в обеих линейках Python, потому что экземпляр автоматически передается методу, который не имеет аргумента для его получения:

```
Spam.printNumInstances()  # Терпит неудачу в Python 2.X, работает в Python 3.X
instance.printNumInstances()    # Терпит неудачу в Python 2.X и 3.X
                                #   (если не является статическим)
```

Если у вас есть возможность использовать Python 3.X и вызывать методы без аргумента `self` только через классы, тогда вы уже имеете в своем распоряжении средство статических методов. Тем не менее, чтобы позволить вызывать методы без `self` через классы в Python 2.X, а также через экземпляры в Python 2.X и 3.X, вам придется либо адаптировать свои проектные решения, либо быть в состоянии как-нибудь пометить такие методы как особые. Давайте рассмотрим оба варианта по очереди.

Альтернативы для статических методов

Не считая пометки метода без аргумента `self` как особого, временами вы можете добиться тех же результатов с помощью других кодовых структур. Скажем, если вы всего лишь хотите вызывать функции, которые обращаются к членам класса без экземпляра, то вероятно простейшей идеей будет применение нормальных функций за пределами класса, а не методов класса. Тогда экземпляр в вызове не ожидается. Вот измененный код, который работает одинаково в Python 3.X и 2.X:

```
def printNumInstances():
    print("Number of instances created: %s" % Spam.numInstances)
          # Количество созданных экземпляров

class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

C:\code> c:\python37\python
>>> import spam
>>> a = spam.Spam()
>>> b = spam.Spam()
>>> c = spam.Spam()
>>> spam.printNumInstances()   # Но функция может находиться слишком далеко
Number of instances created: 3 # И ее невозможно изменить через наследование
>>> spam.Spam.numInstances
3
```

Поскольку имя класса доступно простой функции как глобальная переменная, код работает нормально. Кроме того, обратите внимание, что имя функции становится глобальным, но только для этого одного модуля; оно не конфликтует с именами в других файлах программы.

До появления в Python статических методов такая структура была основной рекомендацией. Из-за того, что в Python уже предлагаются модули в качестве инструмента для разбиения пространств имен на части, можно утверждать, что обычно нет никакой необходимости упаковывать функции в классы, если только они не реализуют поведение объектов. Простые функции внутри модулей вроде показанной здесь делают большинство из того, что могли бы делать методы класса, лишенные экземпляров, и они ассоциированы с классом, т.к. находятся в том же самом модуле.

К сожалению, этот подход все еще далек от идеала. С одной стороны, он добавляет к области видимости файле дополнительное имя, которое используется только для

обработки одиночного класса. С другой стороны, по структуре функция гораздо менее прямо связана с классом; фактически ее определение может занимать сотни строк. Вероятно хуже то, что простые функции подобного рода не могут настраиваться посредством наследования, потому что они существуют вне пространства имен класса: подклассы не в состоянии замещать или расширять функции путем их переопределения.

Мы могли бы попытаться заставить рассматриваемый пример работать в нейтральной к версиям манере, применяя нормальный метод и всегда вызывая его через экземпляр, как обычно:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances(self):
        print("Number of instances created: %s" % Spam.numInstances)
            # Количество созданных экземпляров
C:\code> c:\python37\python
>>> from spam import Spam
>>> a, b, c = Spam(), Spam(), Spam()
>>> a.printNumInstances()
Number of instances created: 3
>>> Spam.printNumInstances(a)
Number of instances created: 3
>>> Spam().printNumInstances()  # Но извлечение счетчика изменяет сам счетчик!
Number of instances created: 4
```

К сожалению, как упоминалось ранее, такой подход совершенно неработоспособен, если нет доступного экземпляра, а создание экземпляра изменяет данные класса, как подтверждает последняя строка выше. Более удачное решение предусматривает пометку метода внутри класса как не требующего экземпляра. В следующем разделе будет показано, каким образом.

Использование статических методов и методов класса

На сегодняшний день существует еще один вариант создания простых функций, ассоциированных с классом, которые могут вызываться либо через класс, либо через его экземпляры. Начиная с версии Python 2.2, мы можем реализовывать классы со статическими методами и методами классов, при вызове не требующих передачи аргумента экземпляра. Для обозначения таких методов в классах вызываются встроенные функции `staticmethod` и `classmethod`, как упоминалось ранее во время обсуждения классов нового стиля. Обе они помечают объект функции как особый, т.е. не требующий передачи экземпляра в случае статического метода и требующий указания аргумента класса в случае метода класса. Например, вот содержимое файла `both-methods.py` (который унифицирует вывод посредством списков в Python 2.X и 3.X, хотя отображение все же слегка отличается для классических классов Python 2.X):

```
# Файл bothmethods.py
class Methods:
    def imeth(self, x):     # Нормальный метод экземпляра: передается self
        print([self, x])

    def smeth(x):           # Статический метод: экземпляр не передается
        print([x])

    def cmeth(cls, x):      # Метод класса: получает класс, а не экземпляр
        print([cls, x])
```

```
smeth = staticmethod(smeth)  # Делает smeth статическим методов (или @: впереди)
cmeth = classmethod(cmeth)  # Делает cmeth методом класса (или @: впереди)
```

Обратите внимание, что последние два оператора присваивания просто повторно присваивают (заново привязывают) имена методов smeth и cmeth. Атрибуты создаются и изменяются любым присваиванием в операторе class, так что финальные операторы присваивания всего лишь переопределяют присваивания, сделанные ранее операторами def. Как вскоре будет показано, специальный синтаксис @ здесь работает в качестве альтернативы в точности как для свойств, но пока не несет особо много смысла, если сначала не разобраться с формой присваивания, которую он автоматизирует.

Формально язык Python теперь поддерживает три вида методов, относящихся к классам, с отличающимися протоколами аргументов:

- методы экземпляров, которым передается объект экземпляра self (стандарт);
- статические методы, которым не передается какой-то дополнительный объект (через staticmethod);
- методы классов, которым передается объект класса (через classmethod, что присуще метаклассам).

Кроме того, модель в Python 3.X расширяется, разрешая простым функциям в классе исполнять роль статических методов без добавочного протокола, когда они вызываются только через объект класса. Несмотря на свое имя, модель bothmethods.py иллюстрирует все три вида методов, поэтому давайте рассмотрим их по очереди.

Методы экземпляров представляют собой нормальный и стандартный случай, который неоднократно встречался в книге. Метод экземпляра должен всегда вызываться с объектом экземпляра. Когда вы вызываете его через *экземпляр*, интерпретатор Python автоматически передает объект экземпляра в первом (крайнем слева) аргументе; при вызове метода экземпляра через *класс* экземпляр потребуется передавать вручную:

```
>>> from bothmethods import Methods    # Нормальные методы экземпляров
>>> obj = Methods()                    # Допускает вызов через экземпляр или класс
>>> obj.imeth(1)
[<bothmethods.Methods object at 0x0000000002A15710>, 1]
>>> Methods.imeth(obj, 2)
[<bothmethods.Methods object at 0x0000000002A15710>, 2]
```

По контрасту *статические методы* вызываются без аргумента экземпляра. В отличие от простых функций за пределами класса их имена являются локальными в рамках областей видимости классов, где они определены, и могут просматриваться процедурой наследования. Функции без экземпляров могут нормально вызываться через класс в Python 3.X, но по умолчанию никогда в Python 2.X. Использование встроенной функции staticmethod позволяет методам подобного рода вызываться также через экземпляр в Python 3.X и через класс и экземпляр в Python 2.X (т.е. первый из приведенных далее вызовов работает в Python 3.X без staticmethod, но второй — нет):

```
>>> Methods.smeth(3)         # Статический метод: вызов через класс
[3]                          # Экземпляр не передается и не ожидается
>>> obj.smeth(4)             # Статический метод: вызов через экземпляр
[4]                          # Экземпляр не передается
```

Методы класса похожи, но интерпретатор Python автоматически передает методу класса в первом (крайнем слева) аргументе класс (не экземпляр) независимо от того, вызывается он через класс или через экземпляр:

```
>>> Methods.cmeth(5)              # Метод класса: вызывается через класс
[<class 'bothmethods.Methods'>, 5]    # Превращается в cmeth(Methods, 5)
>>> obj.cmeth(6)                  # Метод класса: вызывается через экземпляр
[<class 'bothmethods.Methods'>, 6]    # Превращается в cmeth(Methods, 6)
```

В главе 40 обнаружится, что *методы метаклассов* — уникальный, расширенный и формально отличающийся тип методов — ведут себя подобно явно объявленным методам класса, которые исследуются в настоящей главе.

Подсчет экземпляров с помощью статических методов

Теперь, имея в своем распоряжении встроенные функции, реализуем эквивалентный статический метод для подсчета экземпляров — он помечается как особый, так что ему автоматически экземпляр не передается:

```
class Spam:
    numInstances = 0    # Использование статического метода для данных класса
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print("Number of instances: %s" % Spam.numInstances)
            # Количество экземпляров
    printNumInstances = staticmethod(printNumInstances)
```

За счет применения встроенной функции `staticmethod` в коде становится возможным вызов метода без `self` через класс или любой его экземпляр в Python 2.X и 3.X:

```
>>> from spam_static import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()        # Вызывается как простая функция
Number of instances: 3
>>> a.printNumInstances()           # Аргумент экземпляра не передается
Number of instances: 3
```

По сравнению с простым вынесением `printNumInstances` за пределы класса, как было предписано ранее, эта версия требует дополнительного вызова `staticmethod` (или строки `@`, которую мы увидим позже). Однако она также локализует имя функции в области видимости класса (потому оно не конфликтует с другими именами в модуле), перемещает код поближе к месту его использования (внутри оператора `class`) и позволяет подклассам *настраивать* статический метод посредством наследования — более удобный и мощный подход, чем импортирование функций из файлов, в которых находится код суперклассов. Ниже приведен соответствующий подкласс и тестовый сеанс (после изменения файла обязательно запустите новый сеанс, чтобы оператор `from` загрузил новую версию):

```
class Sub(Spam):
    def printNumInstances():          # Переопределение статического метода
        print("Extra stuff...")       # С вызовом первоначального метода
        Spam.printNumInstances()
    printNumInstances = staticmethod(printNumInstances)

>>> from spam_static import Spam, Sub
>>> a = Sub()
>>> b = Sub()
```

```
>>> a.printNumInstances()          # Вызов из экземпляра подкласса
Extra stuff...
Number of instances: 2
>>> Sub.printNumInstances()         # Вызов из самого подкласса
Extra stuff...
Number of instances: 2
>>> Spam.printNumInstances()        # Вызов первоначальной версии
Number of instances: 2
```

Кроме того, классы могут наследовать статический метод, не переопределяя его — он запускается без экземпляра независимо от места определения в дереве классов:

```
>>> class Other(Spam): pass          # Наследование статического метода
>>> c = Other()
>>> c.printNumInstances()
Number of instances: 3
```

Обратите внимание, что это также увеличивает счетчик экземпляров *суперкласса*, поскольку его конструктор наследуется и запускается — поведение, которое начнет проясняться в следующем разделе.

Подсчет экземпляров с помощью методов классов

Интересно отметить, что *метод класса* способен здесь делать похожую работу — показанная ниже версия класса обладает тем же самым поведением, что и рассмотренная ранее версия со статическим методом, но в ней применяется метод класса, который получает класс экземпляра в своем первом аргументе. Вместо жесткого кодирования имени класса метод класса обобщенным образом использует автоматически передаваемый объект класса:

```
class Spam:
    numInstances = 0  # Вместо статического метода используется метод класса
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s" % cls.numInstances)
              # Количество экземпляров
    printNumInstances = classmethod(printNumInstances)
```

Данная версия класса применяется аналогично предшествующим версиям, но ее метод printNumInstances при вызове через класс и через экземпляр получает класс Spam, а не экземпляр:

```
>>> from spam_class import Spam
>>> a, b = Spam(), Spam()
>>> a.printNumInstances()        # В первом аргументе передается класс
Number of instances: 2
>>> Spam.printNumInstances()     # Также в первом аргументе передается класс
Number of instances: 2
```

Тем не менее, при использовании методов классов помните о том, что они получают наиболее специфический (т.е. *самый нижний*) класс объекта, на котором производился вызов. В итоге возникают тонкие последствия при попытке обновления данных класса через переданный класс. Например, создадим в модуле spam_class.py подкласс для настройки, как поступали ранее, дополним метод Spam.printNumInstances, чтобы он также отображал свой аргумент cls, и запустим новый тестовый сеанс:

```
class Spam:
    numInstances = 0                    # Отслеживание переданного класса
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s %s" % (cls.numInstances, cls))
              # Количество экземпляров
    printNumInstances = classmethod(printNumInstances)

class Sub(Spam):
    def printNumInstances(cls):         # Переопределение метода класса
        print("Extra stuff...", cls)    # С вызовом первоначального метода
        Spam.printNumInstances()
    printNumInstances = classmethod(printNumInstances)

class Other(Spam): pass                 # Наследование метода класса
```

Всякий раз, когда запускается метод класса, передается самый нижний класс, даже для подклассов, которые не имеют собственных методов классов:

```
>>> from spam_class import Spam, Sub, Other
>>> x = Sub()
>>> y = Spam()
>>> x.printNumInstances()               # Вызов из экземпляра подкласса
Extra stuff... <class 'spam_class.Sub'>
Number of instances: 2 <class 'spam_class.Spam'>
>>> Sub.printNumInstances()             # Вызов из самого подкласса
Extra stuff... <class 'spam_class.Sub'>
Number of instances: 2 <class 'spam_class.Spam'>
>>> y.printNumInstances()               # Вызов из экземпляра суперкласса
Number of instances: 2 <class 'spam_class.Spam'>
```

Здесь в первом вызове обращение к методу класса производится через экземпляр подкласса Sub и интерпретатор Python передает методу класса самый нижний класс Sub. В данном случае все в порядке — поскольку переопределенная версия метода в Sub явно вызывает версию метода из суперкласса Spam, метод суперкласса Spam получает в своем первом аргументе собственный класс. Но взгляните, что происходит для объекта, который наследует метод класса без изменений:

```
>>> z = Other()                 # Вызов из экземпляра более низкого подкласса
>>> z.printNumInstances()
Number of instances: 3 <class 'spam_class.Other'>
```

Последний вызов передает Other методу класса Spam. В данном примере это работает из-за того, что *извлечение* счетчика находит его в Spam благодаря наследованию. Однако если бы метод попытался *присвоить* данные переданному классу, тогда он обновил бы Other, а не Spam! В таком специфическом случае в Spam вероятно лучше жестко закодировать собственное имя класса, чтобы обновлять свои данные, если есть намерение подсчитывать экземпляры также и всех подклассов, не полагаясь на передаваемый аргумент класса.

Подсчет экземпляров по классам с помощью методов классов

На самом деле из-за того, что методы классов всегда получают *самый нижний* класс в дереве наследования:

- статические методы и явные имена классов могут оказаться лучшим решением при обработке данных, локальных для класса;
- методы классов могут лучше подойти для обработки данных, отличающихся для каждого класса в иерархии.

Скажем, в коде, нуждающемся в управлении счетчиками экземпляров *по классам*, может быть лучше задействовать методы класса. В приведенном далее суперклассе верхнего уровня метод класса используется для управления информацией состояния, которая хранится в классах дерева и варьируется для каждого класса. По духу это похоже на способ, которым методы экземпляров управляют информацией состояния, отличающейся для каждого экземпляра класса:

```
class Spam:
    numInstances = 0
    def count(cls):                 # Счетчики экземпляров по классам
        cls.numInstances += 1       # cls - самый нижний класс над экземпляром
    def __init__(self):
        self.count()                # Передает self.__class__ методу count
    count = classmethod(count)

class Sub(Spam):
    numInstances = 0
    def __init__(self):             # Переопределяет __init__
        Spam.__init__(self)

class Other(Spam):                  # Наследует __init__
    numInstances = 0
```

```
>>> from spam_class2 import Spam, Sub, Other
>>> x = Spam()
>>> y1, y2 = Sub(), Sub()
>>> z1, z2, z3 = Other(), Other(), Other()
>>> x.numInstances, y1.numInstances, z1.numInstances # Данные для каждого класса!
(1, 2, 3)
>>> Spam.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)
```

Статические методы и методы классов исполняют дополнительные расширенные роли, которые в главе не рассматриваются; ищите описание добавочных сценариев на других ресурсах. Тем не менее, в последних версиях Python обозначение статических методов и методов классов стало даже еще проще с появлением синтаксиса *декорирования функций* — способа применения одной функции к другой. Роли этого синтаксиса выходят далеко за рамки сценария использования статических методов, который был первоначальной мотивацией его введения. Синтаксис декорирования функций также позволяет дополнять *классы* в Python 2.X и 3.X, чтобы инициализировать данные, подобные счетчику numInstances в последнем примере. Прием будет представлен в следующем разделе.

В качестве постскриптума по видам методов Python: обязательно ознакомьтесь с описанием *методов метаклассов* в главе 40 — из-за того, что такие методы предназначены для обработки *класса*, который является экземпляром метакласса, они оказываются очень похожими на определенные здесь методы класса, но не требуют объявления classmethod и применяются только к области метаклассов.

Декораторы и метаклассы: часть 1

Поскольку прием с вызовами `staticmethod` и `classmethod`, описанный в предыдущем разделе, некоторым поначалу кажется непонятным, в конечном итоге было добавлено средство для упрощения работы. *Декораторы* Python, аналогичные понятию и синтаксису аннотаций в Java, решают эту специфическую потребность и предоставляют универсальный инструмент для добавления логики, которая управляет функциями и классами или позже обращается к ним.

Средство называется "декорацией", но более конкретно оно является просто способом запуска добавочных шагов обработки на стадии определения функций и классов с помощью явного синтаксиса. Доступны две его разновидности.

- Декораторы функций, появившиеся в Python 2.4, дополняют определения функций. Они задают специальные режимы работы для простых функций и методов классов путем их помещения в добавочный уровень логики, реализованной как еще одна функция, которая обычно называется метафункцией.
- Декораторы классов, добавленные позже в версиях Python 2.6 и 3.0, дополняют определения классов. Они делают то же самое для классов, добавляя поддержку управления целыми объектами и их интерфейсами. Возможно, будучи более простыми, декораторы классов часто пересекаются в исполняемых ролях с метаклассами.

В свою очередь *декораторы функций* представляют собой очень универсальные инструменты: они удобны для добавления к функциям многих видов логики помимо сценариев со статическими методами и методами классов. Скажем, декораторы функций могут использоваться для дополнения функций кодом, который регистрирует в журнале обращения к ним, проверяет типы передаваемых аргументов на стадии отладки и т.д. Декораторы функций могут применяться для управления либо самими функциями, либо их вызовами в более позднее время. В последнем режиме декораторы функций похожи на паттерн проектирования с *делегированием*, исследованный в главе 31, но они предназначены для дополнения вызова специфической функции или метода, а не целого интерфейса объекта.

В Python предлагается несколько встроенных декораторов функций для операций вроде пометки статических методов и методов классов и определения свойств (ранее уже кратко упоминалось, что встроенная функция `property` автоматически работает как декоратор), но программисты также могут реализовывать собственные декораторы. Хотя определяемые пользователем декораторы функций строго не привязаны к классам, они часто записываются в виде классов, чтобы сохранить исходные функции для последующей отправки вместе с другими данными в качестве информации состояния.

Решение оказалось настолько удобной привязкой, что в версиях Python 2.6, 2.7 и 3.X оно было расширено — *декораторы классов* дополняют классы и более непосредственно привязаны к модели классов. Подобно декораторам функций декораторы классов могут управлять самими классами или вызовами для создания экземпляров, нередко используя во втором режиме *делегирование*. Как выяснится, их роли также часто пересекаются с ролями *метаклассов*; в таком случае более новые декораторы классов способны предложить легковесный способ достижения тех же самых целей.

Основы декораторов функций

Синтаксически декораторы функций представляют собой разновидность объявления во время выполнения относительно следующих за ними функций. Декоратор функции записывается в собственной строке перед оператором def, определяющим функцию или метод. Он состоит из символа @, за которым следует то, что мы называем *метафункцией* – функция (или другой вызываемый объект), управляющая другой функцией. Например, начиная с Python 2.4, статические методы могут записываться с помощью синтаксиса декораторов:

```
class C:
   @staticmethod                    # Синтаксис декорирования функций
   def meth():
       ...
```

Внутренне такой синтаксис имеет тот же самый эффект, что и показанный ниже – передача функции в staticmethod и присваивание результата исходному имени:

```
class C:
   def meth():
       ...
   meth = staticmethod(meth)    # Эквивалент в виде повторной привязки имени
```

Декорирование *повторно привязывает* имя метода к результату декоратора. В итоге последующее обращение к имени функции метода фактически первым запускает результат его декоратора staticmethod. Поскольку декоратор способен возвращать объект любого рода, это позволяет декоратору вставлять уровень логики, подлежащий выполнению при каждом вызове. Функция декоратора вольна возвращать либо саму исходную функцию, либо новый *промежуточный* объект, который хранит переданную декоратору исходную функцию для непрямого вызова после выполнения уровня дополнительной логики.

Благодаря декорированию появляется лучший способ реализации в Python 2.X или 3.X нашего примера со статическим методом из предыдущего раздела:

```
class Spam:
   numInstances = 0
   def __init__(self):
      Spam.numInstances = Spam.numInstances + 1

   @staticmethod
   def printNumInstances():
      print("Number of instances created: %s" % Spam.numInstances)
           # Количество созданных экземпляров

>>> from spam_static_deco import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()    # Работают вызовы из классов и экземпляров
Number of instances created: 3
>>> a.printNumInstances()
Number of instances created: 3
```

Из-за того, что встроенные функции classmethod и property также принимают и возвращают функции, их можно аналогичным образом применять в качестве декораторов – как в следующей модификации предыдущего файла bothmethods.py:

```
# Файл bothmethods_decorators.py

class Methods(object):    # Для методов установки свойств в Python 2.X
                          #   необходим object
    def imeth(self, x):   # Нормальный метод экземпляра: передается self
        print([self, x])

    @staticmethod
    def smeth(x):         # Статический метод: экземпляр не передается
        print([x])

    @classmethod
    def cmeth(cls, x):    # Метод класса: получает класс, не экземпляр
        print([cls, x])

    @property             # Свойство: значение вычисляется при извлечении
    def name(self):
        return 'Bob ' + self.__class__.__name__
```

```
>>> from bothmethods_decorators import Methods
>>> obj = Methods()
>>> obj.imeth(1)
[<bothmethods_decorators.Methods object at 0x0000000002A256A0>, 1]
>>> obj.smeth(2)
[2]
>>> obj.cmeth(3)
[<class 'bothmethods_decorators.Methods'>, 3]
>>> obj.name
'Bob Methods'
```

Имейте в виду, что staticmethod и родственные инструменты по-прежнему являются функциями; они могут использоваться в синтаксисе декорирования просто за счет приема функции в качестве аргумента и возвращения вызываемого объекта, к которому может быть повторно привязана исходная функция. В действительности подобным образом можно применять любую такую функцию — даже реализуемые нами самостоятельно функции, определяемые пользователем, как объясняется в следующем разделе.

Первый взгляд на декораторы функций, определяемые пользователем

Хотя Python предлагает несколько встроенных функций, которые могут использоваться как декораторы, мы также можем создавать собственные декораторы. Из-за их широкой практичности мы собираемся посвятить теме написания декораторов отдельную главу в последней части книги. Однако в качестве краткого примера давайте взглянем на простой определяемый пользователем декоратор в работе.

Вспомните из главы 30, что метод перегрузки операций `__call__` реализует интерфейс вызова функций для экземпляров класса. В следующем коде такой прием применяется для определения *промежуточного* класса вызовов, который сохраняет декорированную функцию в экземпляре и перехватывает обращения к исходному имени. Поскольку это класс, он также имеет информацию о состоянии — счетчик сделанных вызовов:

```
class tracer:
    def __init__(self, func):    # Запоминает исходный начальный счетчик
        self.calls = 0
        self.func = func
```

```
    def __call__(self, *args): # При последующих вызовах: добавляет логику,
                               #   запускает оригинал
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args)

@tracer                        # То же самое, что и spam = tracer(spam)
def spam(a, b, c):             # Помещает spam внутрь объекта декоратора
    return a + b + c

print(spam(1, 2, 3)) # На самом деле обращается к объекту-оболочке tracer
print(spam('a', 'b', 'c'))  # Вызывается метод __call__ из класса
```

Так как функция spam выполняется через декоратор tracer, когда происходит обращение к исходному имени spam, она фактически запускает метод `__call__` из класса. Данный метод подсчитывает и регистрирует вызов, после чего передает его исходной функции внутри оболочки. Обратите внимание на использование синтаксиса аргументов *args для упаковки и распаковки передаваемых аргументов; благодаря упомянутому синтаксису декоратор может применяться для помещения в оболочку любой функции с любым количеством позиционных аргументов.

Совокупный эффект снова заключается в добавлении уровня логики к исходной функции spam. Ниже показан вывод сценария в Python 3.X и 2.X — первая строка относится к классу tracer, а вторая дает результирующее значение самой функции spam:

```
c:\code> python tracer1.py
call 1 to spam
6
call 2 to spam
abc
```

Отследите код примера, чтобы лучше его понимать. Как таковой, созданный декоратор работает для любой функции, принимающей позиционные аргументы, но не поддерживает *ключевые* аргументы и не может декорировать функции *методов* уровня класса (короче говоря, в таком случае его методу `__call__` будет передаваться только экземпляр tracer). Как будет показано в части VIII, существуют разнообразные способы написания декораторов функций, в том числе вложенные операторы def; некоторые альтернативы лучше подходят для методов, чем представленная здесь версия.

Скажем, за счет использования вложенных функций с объемлющими областями видимости для состояния вместо экземпляров вызываемого класса с атрибутами декораторы функций часто становятся более широко применимыми также и к методам уровня классов. Мы отложим изложение дальнейших деталей, но далее приведен краткий пример *замыкания*, основанного на этой кодовой модели; оно использует атрибуты функции для состояния счетчика ради совместимости, но в Python 3.X могло бы задействовать переменные и nonlocal:

```
def tracer(func):              # Запоминает оригинал
    def oncall(*args):         # При последующих вызовах
        oncall.calls += 1
        print('call %s to %s' % (oncall.calls, func.__name__))
        return func(*args)
    oncall.calls = 0
    return oncall

class C:
    @tracer
    def spam(self,a, b, c): return a + b + c
```

```
x = C()
print(x.spam(1, 2, 3))
print(x.spam('a', 'b', 'c'))   # Такой же вывод, как у tracer1 (в tracer2.py)
```

Первый взгляд на декораторы классов и метаклассы

Декораторы функций оказались настолько полезными, что в версиях Python 2.6 и 3.0 модель была расширена, сделав возможным применение декораторов также к классам и функциям. *Декораторы классов* похожи на декораторы функций, но выполняются в конце оператора `class` для повторной привязки имени класса к вызываемому объекту. По существу они могут использоваться либо для управления классами сразу после их создания, либо для вставки уровня логики оболочки, чтобы управлять экземплярами, когда они создаются в более позднее время. Условно кодовая структура:

```
def decorator(aClass): ...

@decorator                  # Синтаксис декоратора класса
class C: ...
```

отображается на следующий эквивалент:

```
def decorator(aClass): ...

class C: ...                # Эквивалент в виде повторной привязки имени
C = decorator(C)
```

Декоратор класса способен дополнять сам класс или возвращать *промежуточный* объект, который перехватывает последующие вызовы для создания экземпляров. Например, мы могли бы применять такую привязку в коде из раздела "Подсчет экземпляров по классам с помощью методов классов" ранее в главе для автоматического дополнения классов счетчиками экземпляров и любыми другими требующимися данными:

```
def count(aClass):
    aClass.numInstances = 0
    return aClass           # Возвращает сам класс, а не оболочку

@count
class Spam: ...             # То же самое, что и Spam = count(Spam)

@count
class Sub(Spam): ...        # numInstances = 0 здесь не требуется

@count
class Other(Spam): ...
```

На самом деле в том виде, как есть, данный декоратор может применяться к классам *или* к функциям — он успешно возвращает объект, определяемый в любом из двух контекстов, после инициализации атрибута объекта:

```
@count
def spam(): pass            # Подобно spam = count(spam)

@count
class Other: pass           # Подобно Other = count(Other)

spam.numInstances           # Оба устанавливаются в 0
Other.numInstances
```

Хотя декоратор count управляет функцией или самим классом, позже в книге будет показано, что декораторы классов могут также управлять целым *интерфейсом* объекта. Для этого он перехватывает вызовы создания экземпляров и помещает новый объект

в *промежуточный* объект, вводящий в действие инструменты доступа к атрибутам для будущих запросов – многоуровневая кодовая методика, которую мы будем использовать при реализации защиты атрибутов в главе 39. Вот предварительный набросок модели:

```
def decorator(cls):                  # При декорировании @
    class Proxy:
        def __init__(self, *args):    # При создании экземпляров: создать cls
            self.wrapped = cls(*args)
        def __getattr__(self, name):  # При извлечении атрибутов:
                                      #   выполняются добавочные операции
            return getattr(self.wrapped, name)
    return Proxy

@decorator
class C: ...      # Подобно C = decorator(C)
X = C()           # Создает объект Proxy, являющийся оболочкой C,
                  #   и в будущем перехватывает X.attr
```

Кратко упомянутые ранее *метаклассы* являются похожими расширенными инструментами на основе классов, чьи роли зачастую пересекаются с ролями декораторов классов. Они предоставляют альтернативную модель, которая направляет операции создания объектов класса подклассу класса `type` верхнего уровня в заключение оператора `class`:

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
    ...добавочная логика + создание объектов класса через обращение к type...

class C(metaclass=Meta):
  ...моя операция создания направляется Meta...  # Подобно C = Meta('C', (), {...})
```

В Python 2.X результат такой же, но код отличается – вместо ключевого аргумента в заголовке `class` в нем применяется атрибут класса:

```
class C:
    __metaclass__ = Meta
    ...моя операция создания направляется Meta...
```

В обеих линейках Python обращается к метаклассу класса для создания нового объекта класса, передавая ему данные, которые определены во время выполнения оператора `class`; в Python 2.X метакласс просто по умолчанию вызывает средство создания классического класса:

```
имя_класса = Meta(имя_класса, суперклассы, словарь_атрибутов)
```

Чтобы взять на себя контроль над созданием либо инициализацией нового объекта класса, в метаклассе обычно переопределяется метод `__new__` или `__init__` класса `type`, который в нормальных условиях перехватывает данный вызов. Как и в случае декораторов классов, совокупным эффектом будет определение кода, подлежащего автоматическому запуску на стадии создания объектов класса. Здесь указанный шаг привязывает имя класса к результату вызова определяемого пользователем метакласса. На самом деле метакласс вообще не обязан быть классом – такая возможность, которую мы исследуем позже, размывает отличие между метаклассами и декораторами и даже позволяет их квалифицировать как функционально эквивалентные во многих ролях.

Обе схемы, декораторы классов и метаклассы, способны дополнять класс или возвращать произвольный объект для его замены – протокол с практически безграничными возможностями настройки на основе классов. Как выяснится далее, метаклассы

могут также определять *методы*, которые обрабатывают классы экземпляров, а не их нормальные экземпляры. Эта подобная методам классов методика может эмулироваться посредством методов и данных в промежуточных объектах декораторов классов или даже декоратором классов, который возвращает экземпляр метакласса. Такие трудные для понимания концепции требуют знания фундаментальных понятий, изложенных в главе 40 (и вполне вероятно спокойствия!).

Дополнительные сведения

Естественно, история декораторов и метаклассов гораздо обширнее, чем было показано здесь. Хотя они являются универсальным механизмом, необходимым для ряда пакетов, в реализации *новых* определяемых пользователем декораторов и метаклассов в основном заинтересованы разработчики инструментов, но не прикладные программисты. В связи с этим мы откладываем дальнейшее раскрытие данной темы до последней необязательной части книги:

- в главе 38 более подробно объясняется, каким образом реализовывать свойства с использованием синтаксиса декораторов функций;
- в главе 39 приводятся дополнительные сведения о декораторах, включая более полные примеры;
- в главе 40 рассматриваются метаклассы, а также продолжается история об управлении классами и экземплярами.

Хотя указанные главы охватывают сложные темы, они также дают шанс увидеть Python в работе на более значимых примерах, нежели в других местах книги. А теперь давайте перейдем к обсуждению финальной темы, связанной с классами.

Встроенная функция `super`: для лучшего или для худшего?

До сих пор о встроенной функции `super` упоминалось лишь мимоходом из-за ее относительно редкого и даже сомнительного употребления. Тем не менее, учитывая возросшую за последние годы популярность данного вызова, он заслуживает дополнительного обсуждения. Помимо представления встроенной функции `super` настоящий раздел также служит учебным примером проектирования языка и завершает главу, посвященную настолько многочисленным инструментам, наличие которых в языке написания сценариев вроде Python может показаться необычным.

Некоторые материалы этого раздела ставят под сомнение распространенность инструментов, и я призываю вас самостоятельно оценить любое приводимое в разделе субъективное суждение (мы еще вернемся к затрагиваемым здесь вопросам в конце книги после рассмотрения других расширенных инструментов, таких как метаклассы и дескрипторы). Однако стремительный темп роста языка Python в нынешнее время представляет собой стратегическую точку принятия решения для продвижения его сообщества, и встроенная функция `super` выглядит хорошим показательным примером.

Продолжительные дебаты относительно `super`

Как отмечалось в главах 28 и 29, в Python имеется встроенная функция `super`, которую можно применять для вызова методов суперкласса обобщенным образом, но ее описание откладывалось вплоть до этого момента, что и планировалось. Поскольку

вызов super в типовом коде обладает значительными недостатками, а единственный случай его использования многим кажется неясным и сложным, большинству новичков лучше подойдет применяемая до сих пор традиционная схема вызовов по явным именам. Краткое логическое обоснование данной политики приводилось во врезке "Как насчет super?" в главе 28.

Похоже, в этой области имеются противоречия в самом сообществе Python, весь спектр которых отражен в онлайновых статьях, например, https://fuhm.net/super-harmful. Откровенно говоря, на моих учебных курсах встроенная функция super вызывает наибольший интерес у программистов на Java, приступивших к использованию Python. Причиной является ее концептуальное сходство с инструментом языка Java (многие новые функциональные средства Python в конечном итоге обязаны своим существованием программистам на других языках, которые привнесли свои старые привычки в новую модель). Инструмент super в Python — не то же самое, что super в Java; он по-другому переносится на множественное наследование Python и имеет сценарий применения, выходящий за рамки Java, но с момента его появления ему удалось породить как разногласия, так и неправильное толкование.

Рассмотрение встроенной функции super было отложено в текущем издании (и почти полностью опущено в предшествующих изданиях) из-за значительных проблем. Ее нелегко использовать в Python 2.X, она отличается по форме в линейках Python 2.X и Python 3.X, основана на необычной семантике в Python 3.X, а также плохо сочетается с множественным наследованием Python и перегрузкой операций в типичном коде на Python. Как мы увидим, в определенном коде вызов super фактически может скрывать проблемы и препятствовать более ясному стилю написания кода, обеспечивающему лучший контроль.

В свою очередь вызов super имеет и допустимый сценарий применения — кооперативный режим для устранения конфликтов в деревьях множественного наследования с ромбовидными схемами, который востребован многими новичками. Он требует согласованного и единообразного использования super почти как __slots__, при упорядочении вызовов полагается на довольно-таки неясный алгоритм MRO и предназначен для случая, который в программах на Python гораздо больше считается исключением, чем правилом. В данной роли встроенная функция super похожа на расширенный и основанный на экзотических принципах инструмент, который выходит за рамки интересов большей части аудитории Python и кажется неестественным при достижении целей реальных программ. Кроме того, ожидание его универсального применения выглядит нереалистичным для подавляющего большинства существующего кода на Python.

Из-за всех изложенных факторов в такой вводной книге предпочтение до сих пор отдавалось схеме вызова по явным именам и рекомендовалось поступать подобным образом новичкам. Вам лучше сначала изучить традиционную схему и возможно в целом ее придерживаться вместо того, чтобы использовать добавочный инструмент для особых случаев, который может не работать в ряде контекстов и опирается на загадочную магию в допустимых, но нетипичных ситуациях. Это не просто мнение автора; несмотря на благие намерения сторонников встроенной функции super, по вполне обоснованным причинам в наши дни она не получила широкого признания как "установившаяся практика" в Python.

С другой стороны, как и с другими инструментами, растущее в последние годы применение вызова super в коде Python больше не делает его необязательным для многих программистов на Python — встретив вызов super в первый раз, знайте, что он официально обязателен! Для читателей, которым интересно поэкспериментиро-

вать с super, и для тех, которые вынуждены работать с super, в настоящем разделе предлагается краткий обзор данного инструмента и подоплека его существования, начиная с альтернатив.

Традиционная форма вызова методов суперкласса: переносимая, универсальная

В приводимых примерах предпочтение отдается вызову методов суперкласса, когда это необходимо, за счет явного указания имени суперкласса, потому что такая методика является традиционной в Python, она работает одинаково в Python 2.X и 3.X, а также из-за прочих ограничений и сложностей, связанных с данным вызовом в Python 2.X и 3.X. Как было показано ранее, традиционная схема вызова методов суперкласса с целью их дополнения выглядит следующим образом:

```
>>> class C:                    # В Python 2.X и 3.X
        def act(self):
           print('spam')

>>> class D(C):
        def act(self):
           C.act(self)          # Явное указание имени суперкласса и передача self
           print('eggs')

>>> X = D()
>>> X.act()
spam
eggs
```

Такая форма работает одинаково в Python 2.X и 3.X, соответствует нормальной модели отображения вызовов методов, применяется ко всем видам деревьев наследования и не приводит к запутанному поведению, когда используется перегрузка операций. Чтобы посмотреть, почему отличия имеют значение, давайте займемся сравнением с super.

Базовое использование встроенной функции super и связанные с ней компромиссы

В этом разделе мы представим встроенную функцию super в базовом *режиме одиночного наследования* и выясним ее недостатки в данной роли. Мы обнаружим, что в указанном контексте super работает так, как заявлено, но не сильно отличается от традиционных вызовов, полагается на необычную семантику и нелегко вводится в действие в Python 2.X. Что более важно, как только ваши классы разрастаются для применения множественного наследования, такой режим использования super способен маскировать проблемы в коде и координировать вызовы способами, которые вы могли не ожидать.

Странная семантика: магический посредник в Python 3.X

Встроенная функция super фактически исполняет две намеченных роли. Более экзотическая роль из двух — протоколы кооперативной координации в деревьях множественного наследования с ромбовидными схемами — опирается на порядок поиска MRO в Python 3.X, который был позаимствован из языка Dylan и раскрывается позже в настоящем разделе.

Интересующая нас здесь роль употребляется более широко и чаще запрашивается людьми с опытом работы на языке Java — чтобы получить возможность *обобщенно* ука-

зывать суперклассы в деревьях наследования. Она предназначена для содействия более простому сопровождению кода и избегания набора длинных путей со ссылками на суперклассы в вызовах. Следующий вызов в Python 3.X, по крайней мере, на первый взгляд выглядит успешным для достижения этой цели:

```
>>> class C:  # В Python 3.X (только: см. форму super в Python 2.X далее в главе)
        def act(self):
            print('spam')

>>> class D(C):
        def act(self):
            super().act()         # Обобщенная ссылка на суперкласс, без self
            print('eggs')

>>> X = D()
>>> X.act()
spam
eggs
```

Прием работает и сводит к минимуму модификацию кода — вам не придется обновлять вызов, если суперкласс D изменится в будущем. Тем не менее, одним из самых крупных недостатков данного вызова в Python 3.X является его *зависимость от глубокой магии*. Несмотря на предрасположенность к изменениям, в настоящее время он просматривает стек вызовов в поисках автоматически добавляемого аргумента `self` и суперкласса, после чего объединяет их в специальный *промежуточный объект*, который направляет последующие вызовы версии метода из суперкласса. Если это звучит сложно и странно, то так оно и есть. В действительности такая форма вызова вообще не работает вне контекста метода класса:

```
>>> super                         # "Магический" промежуточный объект,
                                  #  который направляет последующие вызовы
<class 'super'>
>>> super()
SystemError: super(): no arguments
Системная ошибка: super(): отсутствуют аргументы

>>> class E(C):
        def method(self):         # self подразумевается в super...только!
            proxy = super()       # Эта форма не имеет смысла вне метода
            print(proxy)          # Вывод обычно скрытого промежуточного объекта
            proxy.act()  # Аргументы отсутствуют: неявно вызывает метод суперкласса!

>>> E().method()
<super: <class 'E'>, <E object>>
spam
```

На самом деле семантика данного вызова не похожа ни на что другое в Python — это не связанный или несвязанный метод, и он каким-то образом находит аргумент `self` несмотря на его отсутствие в вызове. В деревьях одиночного наследования суперкласс доступен из `self` через путь `self.__class__.__bases__[0]`, но крайне неявная природа вызова затрудняет его выявление и даже не считается с принятой в Python явной политикой аргумента `self`, которая остается справедливой *в любом другом месте*. То есть такой вызов нарушает фундаментальную идиому Python для единственного случая применения. Он также серьезно противоречит существующему издавна в Python правилу проектирования EIBTI (запустите команду `import this`, чтобы получить дополнительные сведения о нем).

Ловушка: наивное добавление множественного наследования

Помимо своей необычной семантики даже в Python 3.X такая роль super применяется наиболее непосредственно к деревьям одиночного наследования и может стать проблематичной, как только классы задействуют множественное наследование с традиционно реализованными классами. Это выглядит крупным ограничением границ использования; из-за полезности *подмешиваемых* классов в Python множественное наследование от разъединенных и независимых суперклассов в реальном коде является, пожалуй, больше нормой, нежели исключением. Вызов super кажется верным путем к катастрофе в классах, реализованных для наивного применения их базового режима без учета гораздо более тонких последствий в деревьях множественного наследования.

Ловушка иллюстрируется в следующем сеансе взаимодействия. Код начинает свое существование, благополучно вводя в действие super в режиме одиночного наследования, чтобы вызывать метод на один уровень выше C:

```
>>> class A:                # В Python 3.X
        def act(self): print('A')
>>> class B:
        def act(self): print('B')

>>> class C(A):
        def act(self):
            super().act()   # super применяется к дереву одиночного наследования
>>> X = C()
>>> X.act()
A
```

Однако если такие классы позже разрастаются для использования более одного суперкласса, то встроенная функция super может стать подверженной ошибкам и даже непригодной к применению. Она не генерирует исключение для деревьев множественного наследования, но будет наивно выбирать только *крайний слева* суперкласс, имеющий выполняемый метод (формально первый согласно MRO), который может быть, а может и не быть тем, что вам нужен:

```
>>> class C(A, B):    # Добавление подмешиваемого класса B с таким же методом
        def act(self):
            super().act()   # Не отказывает при множественном наследовании,
                            #   но выбирает только один метод!
>>> X = C()
>>> X.act()
A

>>> class C(B, A):
        def act(self):
            super().act()   # Если B указывается первым, то A.act()
                            #   больше не запустится!
>>> X = C()
>>> X.act()
B
```

Хуже того, в итоге *молча маскируется* тот факт, что в данном случае вы вероятно должны выбирать суперклассы *явно*, как объяснялось в этой и предыдущей главах. Другими словами, использование super способно скрывать общий источник ошибок в Python — настолько распространенный, что он снова упоминается в затруднениях, связанных с классами, в конце текущей части. Если уж позже у вас может возникнуть необходимость применять прямые вызовы, так почему бы не использовать их сразу?

```
>>> class C(A, B):          # Традиционная форма
        def act(self):      # Вероятно, здесь нужен более явный подход
            A.act(self)     # Такая форма поддерживает одиночное
                            #   и множественное наследование
            B.act(self)     # И работает одинаково в Python 3.X и 2.X
>>> X = C()                 # Так зачем вообще прибегать к особому случаю с super()?
>>> X.act()
A
B
```

Как вскоре будет показано, вы также в состоянии отреагировать на такие ситуации, задействовав вызовы super в *каждом* классе дерева. Но это еще и один из самых крупных недостатков вызова super — зачем помещать его в каждый класс, когда обычно он не нужен и вполне достаточно применения предшествующей более простой традиционной формы в одиночном классе? Особенно в существующем коде и в новом коде, который использует существующий, такое требование касательно super выглядит неестественным, а то и вообще нереалистичным.

Как мы увидим далее, гораздо более тонкий момент связан с тем, что при распространении вызовов super на множественное наследование в подобной манере они могут обращаться к методам не того класса, который ожидался. Вызовы super будут маршрутизироваться в соответствии с порядком MRO, который в зависимости от того, где еще может применяться super, способен инициировать обращение к методу в классе, *вообще не являющемся суперклассом для вызывающего класса* — неявное упорядочение, обеспечивающее интересные сеансы отладки! Если только вы досконально не разберетесь, что встроенная функция super означает после введения множественного наследования, то вероятно лучше не использовать ее и в режиме одиночного наследования.

Такая кодовая ситуация не настолько абстрактна, как может показаться. В книге *Programming Python* (http://www.oreilly.com/catalog/9780596158101) приведен реалистичный пример из *PyMailGUI*. В показанных ниже весьма типичных классах Python множественное наследование применяется для смешивания прикладной логики и оконных инструментов из независимых автономных классов, а потому они обязаны явно обращаться к конструкторам обоих суперклассов с помощью прямых вызовов по имени. Вызов super().__init__() будет выполнять только один конструктор, а добавление super повсюду в разъединенных деревьях классов данного примера потребовало бы большего объема работы, не было бы проще и не имело бы смысла в инструментах, предназначенных для произвольного развертывания в клиентах, которые могут использовать super или нет:

```
class PyMailServerWindow(PyMailServer, windows.MainWindow):
    "Tk с добавочным протоколом и подмешиваемыми методами"
    def __init__(self):
        windows.MainWindow.__init__(self, appname, srvrname)
        PyMailServer.__init__(self)

class PyMailFileWindow(PyMailFile, windows.PopupWindow):
    "Toplevel с добавочным протоколом и подмешиваемыми методами"
    def __init__(self, filename):
        windows.PopupWindow.__init__(self, appname, filename)
        PyMailFile.__init__(self, filename)
```

Важный момент здесь в том, что использование вызова super только для сценариев одиночного наследования, где он применяется наиболее ясно, является потенциальным источником ошибок и путаницы. Кроме того, это также означает необходимость

для программистов запоминать два способа достижения той же самой цели, когда во всех случаях могло быть достаточно только одного способа — явных прямых вызовов.

Другими словами, если вы не уверены в том, что на протяжении всего срока эксплуатации своего программного обеспечения не добавите второй суперкласс к классу в дереве, тогда не можете использовать super в режиме одиночного наследования без понимания и учета его гораздо более сложно устроенной роли в деревьях множественного наследования. Позже мы обсудим второй вариант, но он не является необязательным, если вы вообще вводите в действие super.

С более практической точки зрения также не очевидно, что типовой объем *сопровождения кода*, который предполагается предусмотреть для такой роли super, полностью оправдывает ее присутствие. При программировании на Python имена суперклассов в заголовках редко изменяются; когда они есть, то обычно представляют собой от силы очень небольшое количество обращений к суперклассам, подлежащим обновлению внутри класса. И учтите следующее: если в будущем вы добавите новый суперкласс, где не применяется super (как в предыдущем примере), то вам понадобится либо поместить его внутрь промежуточного объекта адаптера, либо дополнить все вызовы super в своем классе, чтобы так или иначе использовать традиционную схему вызовов по явным именам — задача сопровождения, которая выглядит подходящей, но вероятно более подверженной ошибкам, когда вы больше надеетесь на магию super.

Ограничение: перегрузка операций

Как кратко упоминалось в руководстве по библиотекам Python, встроенная функция super также неполноценно работает при наличии методов перегрузки операций `__X__`. Если вы изучите приведенный ниже код, то заметите, что обращения по прямым именам к методам перегрузки в суперклассе работают нормально, но применение super в выражении не приводит к вызову метода перегрузки суперкласса:

```
>>> class C:                           # В Python 3.X
        def __getitem__(self, ix):     # Метод перегрузки операции индексирования
            print('C index')

>>> class D(C):
        def __getitem__(self, ix):     # Переопределение с целью расширения
            print('D index')
            C.__getitem__(self, ix)    # Традиционная форма вызова работает
            super().__getitem__(ix)    # Прямые вызовы по имени тоже работают
            super()[ix]                # Но операции - нет! (__getattribute__)

>>> X = C()
>>> X[99]
C index
>>> X = D()
>>> X[99]
D index
C index
C index
Traceback (most recent call last):
  File "", line 1, in
  File "", line 6, in __getitem__
TypeError: 'super' object is not subscriptable
Трассировка (самый последний вызов указан последним):
  Файл "", строка 1, в
  Файл "", строка 6, в __getitem__
Ошибка типа: объект super не допускает индексацию
```

Такое поведение объясняется тем же самым изменением в классах нового стиля (и в Python 3.X), которое было описано ранее в главе (см. раздел "Процедура извлечения атрибутов для встроенных операций пропускает экземпляры"). Поскольку промежуточный объект, возвращаемый super, использует __getattribute__ для захвата и маршрутизации последующих вызовов методов, ему не удается перехватывать автоматические обращения к методам __X__, которые инициируются встроенными операциями, включая выражения, т.к. их поиск начинается в классе, а не в экземпляре. Это может показаться менее серьезным, чем ограничение множественного наследования, но в целом операции должны работать так же, как эквивалентные вызовы методов, особенно встроенные операции подобного рода. Отсутствие такой поддержки добавляет еще одно исключение для пользователей super, о котором нужно помнить и противостоять.

Хотя в других языках ситуация может отличаться, в Python аргумент self является явным, подмешивание классов через множественное наследование и перегрузка операций распространены, а имена суперклассов меняются редко. Так как super добавляет к языку странный особый случай, с чуждой семантикой, ограниченной областью действия и сомнительной выгодой, большинству программистов на Python может лучше послужить более широко применимая схема с традиционными вызовами. Несмотря на то что с super тоже связано несколько расширенных приложений, которые мы исследуем позже, они могут быть слишком неясными, чтобы делать их обязательной частью инструментария каждого программиста на Python.

В Python 2.X использование отличается: многословные вызовы

Если вы являетесь пользователем Python 2.X, читающим настоящую книгу по обеим версиям, тогда также должны знать, что методика с super непереносима между линейками Python. Формы super в Python 2.X и Python 3.X отличаются – и не только между классическими классами и классами нового стиля. На самом деле super в Python 2.X представляет собой другой инструмент, который не может запускаться в более простой форме из Python 3.X.

Чтобы данный вызов заработал в Python 2.X, в первую очередь потребуется применять *классы нового стиля*. Но даже тогда вы должны также явно передавать super имя промежуточного класса и self, делая этот вызов настолько сложным и многословным, что в большинстве случаев его вероятно легче вообще избегать и просто явно указывать имя суперкласса в соответствии с традиционной кодовой схемой (ради краткости я предлагаю читателям самостоятельно подумать, что означает изменение собственного имени класса для сопровождения кода при использовании формы super из Python 2.X!):

```
>>> class C(object):            # В Python 2.X: только для классов нового стиля
        def act(self):
            print('spam')

>>> class D(C):
        def act(self):
            super(D, self).act()    # Python 2.X: другой формат вызова -
                                    #   выглядит слишком сложным
            print('eggs')           # D может требовать столько же
                                    #   набора/изменения, сколько и C!

>>> X = D()
>>> X.act()
spam
eggs
```

Хотя для обратной совместимости в Python 3.X можно применять форму вызова из Python 2.X, она слишком громоздко вводится в действие в коде Python 3.X, а более подходящая форма из Python 3.X не пригодна к употреблению в Python 2.X:

```
>>> class D(C):
        def act(self):
            super().act()    # Более простой формат вызова из Python 3.X
                             #   терпит неудачу в Python 2.X
            print('eggs')

>>> X = D()
>>> X.act()
TypeError: super() takes at least 1 argument (0 given)
Ошибка типа: super() принимает, по крайней мере, 1 аргумент (0 предоставлено)
```

С другой стороны, традиционная форма вызова с явными именами классов работает в Python 2.X и с классическими классами, и с классами нового стиля, причем точно так же, как в Python 3.X:

```
>>> class D(C):
        def act(self):
            C.act(self)    # Но традиционная схема работает переносимым образом
            print('eggs') # И в коде Python 2.X часто может оказываться проще

>>> X = D()
>>> X.act()
spam
eggs
```

Так затем использовать методику, которая работает лишь в ограниченных контекстах вместо того, чтобы функционировать во многих других? Несмотря на сложность, в последующих разделах предпринимается попытка обосновать поддержку super.

Положительные стороны super: изменения деревьев и координирование

Показав вам выше недостатки super, я должен также признаться, что у меня возникало искушение применять этот вызов в коде, который всегда запускается под управлением Python 3.X и в котором используется очень длинный путь ссылки на суперклассы через пакет модуля (т.е. главным образом из-за лени, но компактность кода тоже может иметь значение). Справедливости ради отмечу, что вызов super по-прежнему может быть полезным в нескольких сценариях использования, главный из которых заслуживает здесь краткого представления.

- **Изменение деревьев классов во время выполнения.** Когда суперкласс способен изменяться во время выполнения, то невозможно жестко закодировать его имя в выражении вызова, но вызовы можно координировать с помощью super.

  С другой стороны, такой сценарий крайне редко встречается при программировании на Python, а в контексте подобного рода часто могут применяться и другие методики.

- **Кооперативная координация вызовов методов при множественном наследовании.** Когда деревья множественного наследования обязаны координировать одинаково именованные методы во множестве классов, то super может предложить протокол для упорядоченной маршрутизации вызовов.

С другой стороны, дерево классов должно полагаться на упорядочение классов посредством алгоритма MRO – самого по себе сложного инструмента, неестественного для задачи, которую призвана решать программа. Вдобавок в дереве потребуется написать или дополнить каждую версию метода для использования super. Такое координирование нередко может быть реализовано и другими способами (скажем, через состояние экземпляров).

Как обсуждалось ранее, вызов super также может применяться для выбора суперкласса в обобщенной манере, если стандартный порядок MRO имеет смысл, хотя традиционное явное указание имени суперкласса в коде часто предпочтительнее и даже может быть обязательным. Кроме того, даже допустимые сценарии использования super обычно редко встречаются во многих программах на Python – в какой-то мере они выглядят для некоторых как чисто академическое любопытство. Тем не менее, перечисленные выше два сценария чаще всего приводятся в качестве обоснований вызова super, так что давайте кратко рассмотрим каждый из них.

Изменения классов во время выполнения и super

Суперклассы, которые в состоянии изменяться во время выполнения, препятствуют жесткому кодированию их имен в методах подкласса, тогда как super будет успешно динамически просматривать текущий суперкласс. Однако данный сценарий может очень редко встречаться на практике, чтобы служить оправданием самой модели super, и в исключительных случаях, когда он необходим, часто может быть реализован другими способами. В целях иллюстрации ниже динамически изменяется суперкласс класса C за счет модификации кортежа __bases__ подкласса в Python 3.X:

```
>>> class X:
        def m(self): print('X.m')
>>> class Y:
        def m(self): print('Y.m')
>>> class C(X):                    # Сначала унаследовать от X
        def m(self): super().m() # Здесь нельзя жестко закодировать имя класса
>>> i = C()
>>> i.m()
X.m
>>> C.__bases__ = (Y,)            # Изменить суперкласс во время выполнения!
>>> i.m()
Y.m
```

Прием работает (и разделяет цели трансформации поведения с другой глубинной магией, такой как изменение атрибута __class__ экземпляра), но выглядит чрезвычайно редким. Более того, для достижения того же результата могут быть доступны другие способы – вероятно гораздо проще косвенно вызывать метод через значение кортежа текущего суперкласса: бесспорно специальный код, но только для очень особого случая (и возможно не более специализированный, чем неявная маршрутизация посредством MRO):

```
>>> class C(X):
      def m(self): C.__bases__[0].m(self)  # Специальный код для особого случая
>>> i = C()
>>> i.m()
X.m
>>> C.__bases__ = (Y,)                    # Тот же самый результат, но без super()
>>> i.m()
Y.m
```

Учитывая ранее существовавшие альтернативы, лишь один этот сценарий не выглядит оправданием вызова super, хотя в более сложных деревьях следующее логическое обоснование, основанное на порядке MRO дерева вместо физических ссылок на суперклассы, может оказаться также применимым.

Кооперативная координация вызовов методов при множественном наследовании

Второй из указанных ранее сценариев использования является главным обоснованием, обычно приводимым в пользу super, и также позаимствован из других языков программирования (в особенности из Dylan), где он более распространен, чем в типичном коде на Python. Как правило, он применяется к деревьям множественного наследования с ромбовидными схемами, которые обсуждались ранее в главе, и позволяет кооперативным и совместимым классам логично направлять вызовы *методу с тем же самым именем* в рамках множества реализаций классов. Описанный прием при согласованном использовании способен упростить протокол маршрутизации вызовов, особенно для конструкторов, обычно имеющих много реализаций.

В этом режиме каждый вызов super выбирает метод из *следующего за ним класса* в порядке MRO класса объекта self, передаваемого в вызове метода. Такой процесс выбора отдает предпочтение первому классу, следующему за вызывающим классом, который имеет запрошенный атрибут. Порядок MRO был введен ранее; он представляет собой путь, который интерпретатор Python проходит при наследовании в классах нового стиля. Поскольку линейное упорядочение MRO зависит от того, из какого класса был создан объект self, порядок организуемого вызовом super координирования методов может варьироваться от дерева к дереву с посещением каждого класса только раз при условии, что все классы применяют super для координации.

Так как каждый класс участвует в ромбовидной схеме под object в Python 3.X (в классах нового стиля в Python 2.X), приложения super шире, чем можно было ожидать. Фактически в ряде ранних примеров, демонстрировавших недостатки super внутри деревьев множественного наследования, вызов super можно было бы использовать для достижения их целей координации. Тем не менее, для этого вызов super потребуется применять *повсюду* в дереве классов, чтобы гарантировать прохождение по цепочкам вызова методов – довольно значительное требование, удовлетворить которое может быть нелегко в большинстве существующего и нового кода.

Основы: вызов super при кооперативном координировании в действии

Давайте посмотрим, что такая роль super означает в коде. В этом и следующем разделах мы выясним, как работает super, и попутно исследуем связанные с ним компромиссы. Для начала взгляните на приведенные ниже *традиционно* реализованные классы на Python (показанные в несколько сжатом виде ради экономии пространства):

```
>>> class B:
        def __init__(self): print('B.__init__')   # Разъединенные ветви
                                                  #   дерева классов
>>> class C:
        def __init__(self): print('C.__init__')
>>> class D(B, C): pass

>>> x = D()                      # По умолчанию выполняется только крайний слева
B.__init__
```

В данном случае ветви дерева суперклассов *разъединены* (у них отсутствует явный общий предок), так что подклассы, которые их комбинируют, должны делать вызовы через каждый суперкласс по имени – распространенная ситуация с большинством существующего кода на Python, которую super не может решить напрямую без внесения изменений в код:

```
>>> class D(B, C):
        def __init__(self):       # Традиционная форма
            B.__init__(self)      # Обращение к суперклассам по имени
            C.__init__(self)
>>> x = D()
B.__init__
C.__init__
```

Однако в деревьях классов с *ромбовидными* схемами вызовы *по явным именам* могут по умолчанию запускать метод класса верхнего уровня более одного раза, хотя это можно обойти с помощью дополнительных протоколов (например, маркеров состояния в экземпляре):

```
>>> class A:
        def __init__(self): print('A.__init__')
>>> class B(A):
        def __init__(self): print('B.__init__'); A.__init__(self)
>>> class C(A):
        def __init__(self): print('C.__init__'); A.__init__(self)
>>> x = B()
B.__init__
A.__init__
>>> x = C()                       # Каждый суперкласс работает сам по себе
C.__init__
A.__init__

>>> class D(B, C): pass           # По-прежнему выполняется только крайний слева
>>> x = D()
B.__init__
A.__init__

>>> class D(B, C):
        def __init__(self):       # Традиционная форма
            B.__init__(self)      # Обращение к обоим суперклассам по имени
            C.__init__(self)
>>> x = D()                       # Но теперь обращение к A происходит дважды!
B.__init__
A.__init__
C.__init__
A.__init__
```

Напротив, если все классы используют super или снабжаются промежуточными классами, которые обеспечат надлежащее поведение, тогда вызовы методов координируются в соответствии с порядком классов в MRO, так что метод класса верхнего уровня выполняется только один раз:

```
>>> class A:
        def __init__(self): print('A.__init__')
>>> class B(A):
        def __init__(self): print('B.__init__'); super().__init__()
>>> class C(A):
        def __init__(self): print('C.__init__'); super().__init__()
```

```
>>> x = B()     # Выполняется B.__init__,
                #   A - следующий суперкласс в MRO класса B объекта self
B.__init__
A.__init__
>>> x = C()
C.__init__
A.__init__

>>> class D(B, C): pass
>>> x = D()     # Выполняется B.__init__,
                #   C - следующий суперкласс в MRO класса D объекта self!
B.__init__
C.__init__
A.__init__
```

Реальной магией, стоящей за этим, является линейный список MRO, созданный для класса объекта self. Из-за того, что каждый класс в данном списке встречается только раз и вызов super направляет на *следующий* класс в списке, он обеспечивает упорядоченную цепочку вызовов, предусматривающую посещение каждого класса только один раз. Важно отметить, что *следующий* за B класс в MRO отличается в зависимости от класса объекта self — он будет A для экземпляра B, но C для экземпляра D, учитывая порядок выполнения конструкторов:

```
>>> B.__mro__
(<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

Порядок MRO и алгоритм его выстраивания были представлены ранее в главе. За счет выбора следующего класса в последовательности MRO вызов super в методе класса *распространяет* вызов по дереву при условии, что все классы делают то же самое. В таком режиме вызов super совершенно необязательно выберет суперкласс; он выбирает следующий элемент в линеаризованном списке MRO, который может быть *одного уровня* или даже *более низким* родственником в дереве классов заданного экземпляра. Другие примеры пути, которым могло бы следовать координирование super, особенно для неромбовидных схем, приводились в разделе “Отслеживание MRO” ранее в главе.

Предыдущий прием работает и на первый взгляд может даже выглядеть ловким, но область его действия некоторым кажется ограниченной. Большинство программ на Python не полагаются на нюансы деревьев множественного наследования с ромбовидными схемами (на самом деле многие программисты на Python, которых я встречал, даже не знают, что этот термин означает!). Кроме того, вызов super применим наиболее прямо к сценариям одиночного наследования и кооперативной координации вызовов методов при множественном наследовании с ромбовидными схемами, и может выглядеть чрезмерным для разъединенных неромбовидных случаев, где возможно возникнет желание вызывать методы суперклассов избирательно или независимо. Даже кооперативной координацией вызовов методов при множественном наследовании с ромбовидными схемами можно управлять другими способами, которые способны предложить программистам больший контроль, чем автоматическое упорядочение MRO. Тем не менее, объективная оценка этого инструмента требует углубленных исследований.

Ограничение: требование закрепления цепочки вызовов

Вызов super сопровождается сложностями, которые поначалу могут не быть явными и даже выглядеть как особенности. Скажем, поскольку в Python 3.X *все* классы автоматически наследуются от object (и явно классы нового стиля в Python 2.X), упорядочение MRO можно использовать даже в случаях, когда ромбовидная схема только неявная, и автоматически запускать конструкторы в независимых классах:

```
>>> class B:
        def __init__(self): print('B.__init__'); super().__init__()
>>> class C:
        def __init__(self): print('C.__init__'); super().__init__()

>>> x = B()                 # object - подразумеваемый суперкласс в конце MRO
B.__init__
>>> x = C()
C.__init__

>>> class D(B, C): pass  # Наследуется B.__init__, но MRO в B отличается для D
>>> x = D()                 # Выполняется B.__init__, C - следующий суперкласс
                            #   в MRO класса D объекта self!
B.__init__
C.__init__
```

Формально такая модель координирования в целом требует, чтобы вызываемый через super метод существовал, имел одну и ту же сигнатуру аргументов во всем дереве классов и каждое появление метода кроме последнего само должно применять super. Предыдущий пример работает лишь потому, что подразумеваемый суперкласс object в конце списков MRO всех трех классов располагает совместимым методом __init__, который удовлетворяет данным правилам:

```
>>> B.__mro__
(<class '__main__.B'>, <class 'object'>)
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>)
```

Здесь для экземпляра D следующим классом в MRO после B является C, а за ним находится object, чей метод __init__ молча принимает вызов от C и завершает цепочку. Таким образом, метод B вызывает метод C, который заканчивается в версии метода из object, хотя C — не суперкласс для B.

Однако в действительности рассмотренный пример нетипичен — и вероятно даже *удачный*. В большинстве ситуаций подходящие стандартные методы в object не существуют, а потому удовлетворить ожиданиям модели оказывается не так-то легко. Большинству деревьев будет требоваться явный — и возможно добавочный — суперкласс для исполнения закрепляющей роли, которую в примере играл object, чтобы принимать, но не пересылать вызов. Другие деревья могут требовать тщательного проектирования для удовлетворения такого требования. Кроме того, если только интерпретатор Python не позаботится об оптимизации, то вызов стандартных методов object (или другого закрепляющего класса) в конце цепочки также может быть сопряжен с дополнительными *затратами в плане производительности*.

И наоборот, в случаях подобного рода прямые вызовы не связаны ни с добавочными требованиями относительно кода, ни с дополнительными затратами в плане производительности и делают координирование более явным и непосредственным:

```
>>> class B:
        def __init__(self): print('B.__init__')
```

```
>>> class C:
        def __init__(self): print('C.__init__')
>>> class D(B, C):
        def __init__(self): B.__init__(self); C.__init__(self)

>>> x = D()
B.__init__
C.__init__
```

Область охвата: модель "все или ничего"

Также имейте в виду, что традиционные классы, которые не были рассчитаны на использование super в данной роли, не могут напрямую применяться в деревьях кооперативного координирования, потому что они не будут пересылать вызовы по цепочке MRO. Такие классы можно помещать внутрь промежуточных классов, которые содержат в себе исходный объект и добавляют необходимые вызовы super, но этот подход налагает на модель дополнительные кодовые требования и приводит к снижению производительности. Учитывая существование многих миллионов строк кода на Python, где super *не* используется, решение способно нанести крупный ущерб.

Скажем, посмотрите, что происходит, если какой-то один класс терпит неудачу с передачей вызова по цепочке из-за отсутствия super, преждевременно заканчивая всю цепочку вызовов — подобно __slots__ вызов super в целом является средством вида "все или ничего":

```
>>> class B:
        def __init__(self): print('B.__init__'); super().__init__()
>>> class C:
        def __init__(self): print('C.__init__'); super().__init__()
>>> class D(B, C):
        def __init__(self): print('D.__init__'); super().__init__()
>>> X = D()
D.__init__
B.__init__
C.__init__
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>)

# Что, если мы должны использовать класс, который не вызывает super?
>>> class B:
        def __init__(self): print('B.__init__')
>>> class D(B, C):
        def __init__(self): print('D.__init__'); super().__init__()
>>> X = D()
D.__init__
B.__init__     # Это инструмент, работающий в стиле 'все или ничего'...
```

Удовлетворение требования обязательного распространения может оказаться не проще, чем прямые вызовы по именам — о которых вы по-прежнему можете забывать, но которые не нуждаются в том, чтобы быть обязательными для всего кода, задействованного вашими классами. Как уже упоминалось, класс типа B можно приспособить, наследуя его от *промежуточного* класса, который внедряет экземпляры B, но такой подход выглядит неестественным в отношении целей программы, добавляет дополнительный вызов к каждому внутреннему методу, подвержен проблемам классов нового стиля, с которыми мы сталкивались ранее при исследовании промежуточных интерфейсных классов и встроенных функций, и кажется чрезмерным и даже ошеломляющим добавочным кодовым требованием в модели, предназначенной для упрощения кода.

Гибкость: допущения об упорядочении вызовов

Маршрутизация с помощью super также предполагает, что вы действительно намереваетесь передавать вызовы методов через все классы в соответствии с MRO – это может как совпадать с вашими требованиями к *упорядочению вызовов*, так и нет. Например, представьте себе, что независимо от других потребностей упорядочения, связанных с наследованием, показанный ниже код требует, чтобы в ряде контекстов версия заданного метода из класса C выполнялась перед его версией из класса B. Если алгоритм MRO указывает иное, тогда вы возвращаетесь к традиционным вызовам, которые могут конфликтовать с применением super – версия метода из C вызывается дважды:

```
# Что, если требования к упорядочению вызовов отличаются от MRO?

>>> class B:
        def __init__(self): print('B.__init__'); super().__init__()
>>> class C:
        def __init__(self): print('C.__init__'); super().__init__()
>>> class D(B, C):
       def __init__(self): print('D.__init__'); C.__init__(self); B.__init__(self)
>>> X = D()
D.__init__
C.__init__
B.__init__
C.__init__    # Вызов во второй раз...
```

Точно так же, если вы хотите, чтобы некоторые методы *вообще не выполнялись*, то автоматический путь super не будет применим настолько непосредственно, как явные вызовы, и затруднит получение большего контроля над процессом координирования. В реальных программах с многочисленными методами, ресурсами и переменными состояния такие сценарии выглядят вполне правдоподобными. Хотя вы могли бы переупорядочить суперклассы в D для данного метода, это способно нарушить другие ожидания.

Настройка: замещение методов

В качестве связанного замечания: ожидания при глобальном вводе в действие super могут в целом затруднить *замещение* (переопределение) унаследованного метода в отдельно взятом классе. Отказ от передачи вызова выше посредством super – намеренный в этом случае – нормально работает для самого класса, но может нарушить цепочку вызовов в деревьях, куда он подмешивается, препятствуя выполнению методов где-то в других местах деревьев. Взгляните на следующее дерево:

```
>>> class A:
        def method(self): print('A.method'); super().method()
>>> class B(A):
        def method(self): print('B.method'); super().method()
>>> class C:
        def method(self): print('C.method')   # super отсутствует: цепочка
                                              #   должна быть закреплена!
>>> class D(B, C):
        def method(self): print('D.method'); super().method()
>>> X = D()
>>> X.method()
D.method
B.method
A.method   # Автоматически координируется для всех согласно MRO
C.method
```

Замещение методов здесь нарушает работу модели super и возможно возвращает нас обратно к традиционной форме:

```
# Что, если классу необходимо полностью заместить стандартный метод суперкласса?
>>> class B(A):
      def method(self): print('B.method')  # Замещение метода из суперкласса A
>>> class D(B, C):
      def method(self): print('D.method'); super().method()
>>> X = D()
>>> X.method()
D.method
B.method   # Но замещение также нарушает цепочку вызовов...

>>> class D(B, C):
        def method(self): print('D.method'); B.method(self); C.method(self)
>>> D().method()
D.method
B.method
C.method   # Возвращение к явным вызовам...
```

Имеет смысл повторить еще раз: проблема с допущениями в том, что они что-то предполагают! Хотя допущение глобальной маршрутизации может быть разумным для конструкторов, оно также способно конфликтовать с одним из главных принципов ООП – неограниченной *настройкой в подклассах*. Это может означать ограничение использования super только конструкторами, но даже они иногда требуют замещения, что добавляет странное требование особого случая для одного специфического контекста. Инструмент, который может применяться только для определенных категорий методов, ряду пользователей кажется избыточным – и даже паразитным, учитывая привносимую им сложность.

Связанность: приложение для подмешиваемых классов

Тонкость в том, что когда мы говорим, что super выбирает *следующий класс* в MRO, то на самом деле имеем в виду следующий класс в MRO, *который реализует запрошенный метод* – формально производится просмотр вперед до тех пор, пока не будет найден класс с запрошенным именем. Это важно для независимых подмешиваемых классов, которые могут добавляться в произвольные клиентские деревья. Без такого поведения с просмотром вперед подмешиваемые классы вообще не будут работать – они бы прекращали цепочку вызовов произвольных методов из своих клиентов и не могли бы полагаться на собственные вызовы super для работы ожидаемым образом.

Например, в приведенных далее независимых ветвях вызов method из C передается, хотя класс Mixin, следующий в MRO экземпляра C, не определяет метод с таким именем. До тех пор, пока наборы имен методов разъединены, это просто работает – цепочки вызовов способны существовать независимо:

```
# Подмешиваемые классы работают для разъединенных наборов методов
>>> class A:
        def other(self): print('A.other')
>>> class Mixin(A):
        def other(self): print('Mixin.other'); super().other()

>>> class B:
        def method(self): print('B.method')
>>> class C(Mixin, B):
        def method(self): print('C.method'); super().other(); super().method()
```

```
>>> C().method()
C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.Mixin'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'object'>)
```

Подобным образом подмешивание другим способом также не нарушает цепочки вызовов подмешиваемого класса. Например, в показанном ниже взаимодействии, несмотря на то, что B не определяет other при вызове в C, классы делают это позже в MRO. На самом деле цепочки вызовов работают, даже если в одной из ветвей вообще не используется super — до тех пор, пока метод определен где-то впереди в MRO, обращение к нему работает:

```
>>> class C(B, Mixin):
    def method(self): print('C.method'); super().other(); super().method()
>>> C().method()
C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.Mixin'>,
<class '__main__.A'>, <class 'object'>)
```

Сказанное остается справедливым также при наличии ромбовидных схем — разъединенные наборы методов координируются ожидаемым образом, даже когда они не реализованы каждой разъединенной цепочкой, поскольку мы выбираем следующий класс в MRO с нужным методом. В действительности, поскольку в таких случаях MRO содержит те же самые классы, а подкласс всегда появляется перед его суперклассом в MRO, они представляют собой эквивалентные контексты. Например, вызов метода other внутри Mixin в приведенном далее взаимодействии по-прежнему находит его в A, хотя следующим после Mixin классом в MRO является B (вызов method внутри C работает по схожим причинам):

```
# Явные ромбовидные схемы тоже работают
>>> class A:
        def other(self): print('A.other')
>>> class Mixin(A):
        def other(self): print('Mixin.other'); super().other()
>>> class B(A):
        def method(self): print('B.method')
>>> class C(Mixin, B):
        def method(self): print('C.method'); super().other(); super().method()
>>> C().method()
C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.Mixin'>, <class '__main__.B'>,
```

```
<class '__main__.A'>, <class 'object'>)
# Работают и другие порядки подмешивания
>>> class C(B, Mixin):
        def method(self): print('C.method'); super().other(); super().method()

>>> C().method()
C.method
Mixin.other
A.other
B.method

>>> C.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.Mixin'>,
<class '__main__.A'>, <class 'object'>)
```

Тем не менее, мы получаем результат, ничем не отличающийся, но выглядящий гораздо более неявным, чем прямые вызовы по имени, которые в данном случае также работают аналогично независимо от упорядочения суперклассов и от того, есть ромбовидная схема или нет. В этом сценарии мотив полагаться на упорядочение MRO кажется сомнительным, т.к. традиционная форма проще и яснее, к тому же предлагает больше контроля и гибкости:

```
# Но прямые вызовы также здесь работают: явное лучше неявного
>>> class C(Mixin, B):
        def method(self): print('C.method'); Mixin.other(self);
B.method(self)

>>> X = C()
>>> X.method()
C.method
Mixin.other
A.other
B.method
```

Важнее то, что в приведенном примере до сих пор предполагалась разъединенность имен методов в их ветвях; порядок координирования для *одинаково именованных методов* в ромбовидных схемах такого рода может быть гораздо менее случайным. Скажем, в ромбовидной схеме вроде предыдущей не исключено, что клиентский класс мог бы свести на нет намерение вызова super — обращение к method в Mixin ниже работает, запуская версию метода из A, как и ожидалось, *если только он не* подмешан в дерево, которое нарушает цепочку вызовов:

```
# Но для неразъединенных наборов методов: super создает чрезмерно сильную связность
>>> class A:
        def method(self): print('A.method')
>>> class Mixin(A):
        def method(self): print('Mixin.method'); super().method()
>>> Mixin().method()
Mixin.method
A.method

>>> class B(A):
        def method(self): print('B.method')     # super здесь вызовет A после B
>>> class C(Mixin, B):
        def method(self): print('C.method'); super().method()
>>> C().method()
C.method
Mixin.method
B.method                                 # Мы пропускаем A только в этом контексте!
```

Может случиться так, что B не должен переопределять этот метод в любом случае (откровенно говоря, мы можем вторгнуться в проблемы, присущие множественному наследованию в целом), но это *также* не должно нарушать подмешивание. *Прямые вызовы* предоставляют вам больший контроль в подобных ситуациях и позволяют подмешиваемым классам быть независимыми от контекстов применения:

```
# А прямые вызовы - нет: они невосприимчивы к контексту использования
>>> class A:
        def method(self): print('A.method')
>>> class Mixin(A):
        def method(self): print('Mixin.method'); A.method(self)   # C не связан
>>> class C(Mixin, B):
        def method(self): print('C.method'); Mixin.method(self)
>>> C().method()
C.method
Mixin.method
A.method
```

Кстати, делая подмешиваемые классы более *изолированными*, прямые вызовы минимизируют *связность компонентов*, неизменно увеличивающую сложность программ — фундаментальный принцип разработки программного обеспечения, который, похоже, игнорируется изменчивой и специфической к контексту моделью координирования super.

Настройка: ограничения, связанные с одинаковыми аргументами

В качестве финального замечания: вы также должны принимать во внимание последствия использования super, когда *аргументы метода отличаются* от класса к классу. Из-за того, что разработчик класса не может быть уверен в том, какую версию метода вызовет super (все действительно варьируется в зависимости от дерева!), каждая версия метода обычно обязана принимать тот же самый список аргументов или выбирать свои входные данные с помощью анализа обобщенных списков аргументов. Оба подхода предъявляют дополнительные требования к коду. В реалистичных программах такое ограничение фактически может стать настоящим *препятствием* для многих потенциальных приложений super, исключая его применение вообще.

Чтобы выяснить, почему это может иметь значение, вспомните классы для представления сотрудников пиццерии, которые были реализованы в главе 31. В том виде оба подкласса используют прямые вызовы *по именам* для обращения к конструктору суперкласса, автоматически заполняя ожидаемый аргумент salary — логика подкласса, которая лежит в основе подразумеваемого уровня оплаты:

```
>>> class Employee:
        def __init__(self, name, salary):            # Общий суперкласс
            self.name = name
            self.salary = salary
>>> class Chef1(Employee):
        def __init__(self, name):                    # Отличающиеся аргументы
            Employee.__init__(self, name, 50000)     # Координирование
                                                     #  по прямому вызову
>>> class Server1(Employee):
        def __init__(self, name):
            Employee.__init__(self, name, 40000)
>>> bob = Chef1('Bob')
>>> sue = Server1('Sue')
>>> bob.salary, sue.salary
(50000, 40000)
```

Прием работает, но поскольку это дерево одиночного наследования, у нас может возникнуть искушение ввести здесь в действие super для маршрутизации обращений к конструкторам обобщенным образом. Решение подходит для любого из двух подклассов по отдельности, т.к. его MRO содержит сам класс и фактический суперкласс:

```
>>> class Chef2(Employee):
        def __init__(self, name):
          super().__init__(name, 50000)  # Координируется посредством super()

>>> class Server2(Employee):
        def __init__(self, name):
            super().__init__(name, 40000)

>>> bob = Chef2('Bob')
>>> sue = Server2('Sue')
>>> bob.salary, sue.salary
(50000, 40000)
```

Однако взгляните, что происходит, когда сотрудник относится к *обеим* категориям. Из-за того, что конструкторы в дереве имеют отличающиеся списки аргументов, возникает проблема:

```
>>> class TwoJobs(Chef2, Server2): pass

>>> tom = TwoJobs('Tom')
TypeError: __init__() takes 2 positional arguments but 3 were given
Ошибка типа: __init__() принимает 2 позиционных аргумента, но было
предоставлено 3
```

Проблема связана с тем, что вызов super в Chef2 больше не обращается к своему суперклассу Employee, но взамен делает обращение к *родственному* классу того же уровня и следующему в MRO, т.е. Server2. Поскольку конструктор родственного класса имеет список аргументов, отличающийся от списка аргументов в конструкторе настоящего суперкласса, который ожидает только self и name, код перестает работать. Это присуще применению super: из-за того, что MRO может отличаться от дерева к дереву, в разных деревьях возможны вызовы разных версий метода — даже тех, которые вы не в состоянии предвидеть при реализации самого класса:

```
>>> TwoJobs.__mro__
(<class '__main__.TwoJobs'>, <class '__main__.Chef2'>, <class '__main__.Server2'>
<class '__main__.Employee'>, <class 'object'>)

>>> Chef2.__mro__
(<class '__main__.Chef2'>, <class '__main__.Employee'>, <class 'object'>)
```

Напротив, схема прямых вызовов по именам по-прежнему работает, когда классы смешиваются, хотя результаты слегка неоднозначны — сотрудник с объединенными категориями получает оплату крайнего слева суперкласса:

```
>>> class TwoJobs(Chef1, Server1): pass

>>> tom = TwoJobs('Tom')
>>> tom.salary
50000
```

На самом деле в таком случае, вероятно, мы захотим направить вызов классу верхнего уровня с новой заработной платой — модель, которую возможно реализовать с помощью прямых вызовов, но не исключительно посредством super. Кроме того, обращение к Employee напрямую в этом одном классе означает, что наш код использу-

ет *две* методики координирования, когда было бы достаточно только одной — прямых вызовов:

```
>>> class TwoJobs(Chef1, Server1):
        def __init__(self, name): Employee.__init__(self, name, 70000)
>>> tom = TwoJobs('Tom')
>>> tom.salary
70000
>>> class TwoJobs(Chef2, Server2):
        def __init__(self, name): super().__init__(name, 70000)
>>> tom = TwoJobs('Tom')
TypeError: __init__() takes 2 positional arguments but 3 were given
Ошибка типа: __init__() принимает 2 позиционных аргумента, но было
предоставлено 3
```

В целом рассмотренный пример может потребовать повторного проектирования — скажем, вынесения разделяемых частей Chef и Server в подмешиваемые классы без конструктора. Также верно и то, что полиморфизм в общем случае предполагает наличие в методах из *внешнего* интерфейса объекта той же самой сигнатуры аргументов, хотя это не полностью применимо к настройке методов суперкласса — *внутренняя* методика реализации, которая по своей природе должна поддерживать вариацию, особенно в конструкторах.

Но важный момент здесь в том, что поскольку прямые вызовы не делают код зависимым от магического упорядочения, которое может меняться от дерева к дереву, они более непосредственно содействуют гибкости списков аргументов. В более широком смысле сомнительные (или слабые) рабочие характеристики вызова super влияют на замещение методов, связность подмешиваемых классов, упорядочение вызовов и ограничения аргументов, которые должны побудить вас тщательно взвешивать ввод super в эксплуатацию. Даже в режиме одиночного наследования с разрастанием деревьев его потенциальное влияние *в более позднее время* может оказываться существенным.

В итоге три требования вызова super в такой роли становятся источником большинства проблем с удобством в использовании:

- вызываемый посредством super метод должен существовать, что требует добавочного кода, если закрепление отсутствует;
- вызываемый посредством super метод обязан иметь одну и ту же сигнатуру аргументов повсеместно в дереве классов, что снижает гибкость, особенно для методов уровня реализации, подобных конструкторам;
- каждое вхождение метода вызывается super, но последнее должно применять сам вызов super, что затрудняет использование существующего кода, изменяет порядок вызовов и реализацию автономных классов.

Вместе все перечисленное приводит к получению инструмента со значительной сложностью и существенными компромиссами — недостатками, которые заявят о себе в момент, когда код разрастется для включения множественного наследования.

Естественно, могут найтись изящные способы выхода из только что описанных затруднительных положений, касающихся super, но дополнительные шаги по написанию кода могут еще больше нивелировать преимущества вызова super — и в любом случае нам здесь не хватит места. Также существуют альтернативные решения *без* super для ряда задач координирования методов в деревьях множественного наследования с ромбовидными схемами, но по причинам, связанным с пространством, их

придется оставить в качестве упражнения. Когда методы суперклассов вызываются за счет явного указания имен, корневые классы в деревьях множественного наследования с ромбовидными схемами могли бы проверять состояние в экземплярах во избежание двукратного запуска. Это аналогично сложная кодовая схема, которая редко требуется в большинстве кода, но многим кажется не сложнее применения самого вызова `super`.

Сводка по super

Итак, что мы имеем – плохого и хорошего. Как со всеми расширениями Python, вам придется выносить собственное суждение. Я пытался сохранить честное, беспристрастное отношение к обеим сторонам прений, чтобы помочь вам выработать свое решение. Но поскольку вызов `super`:

- отличается по форме в линейках Python 2.X и Python 3.X;
- в Python 3.X опирается на возможно отличающуюся от стиля Pythonic магию и не полностью применим к перегрузке операций или традиционно реализованным деревьям множественного наследования;
- в Python 2.X выглядит настолько многословным в данной предполагаемой роли, что может сделать код более, а не менее сложным;
- заявляет о преимуществах сопровождения, которые при практическом использовании Python могут оказаться больше гипотетическими, нежели реальными;

то даже бывшие программисты на Java должны рассматривать предпочитаемую в книге традиционную методику обращений к суперклассам по явным именам, по меньшей мере, таким же допустимым решением, как `super` в Python – вызов, на определенных уровнях выглядящий необычным и ограниченным ответом на вопрос, который не задавался большинством программистов на Python и не считался важным для большей части истории Python.

Вместе с тем вызов `super` предлагает одно решение сложной задачи координирования одинаково именованных методов в деревьях множественного наследования для программ, где выбрано его *всеобщее* и согласованное применение. Но в том-то и заключается одно из связанных с ним крупных препятствий: вызов `super` требует глобального ввода в действие, чтобы решить проблему, которую большинство программистов вероятно не испытывают. Кроме того, на этом этапе истории Python предложение программистам изменить существующий у них код для достаточно широкого использования вызова `super`, чтобы он стал надежным, представляется совершенно *нереалистичным*.

Тем не менее, возможно главной проблемой данной роли является *сама роль* – координирование одинаково именованных методов в деревьях множественного наследования редко встречается в реальных программах на Python и достаточно неясно, чтобы стать причиной многих разногласий и непонимания. Люди применяют Python не так, как используют C++, Java или Dylan, и уроки, вынесенные из других языков, не обязательно помогут.

Также имейте в виду, что применение `super` делает поведение вашей программы зависимым от алгоритма MRO – процедуры, которая из-за своей сложности была раскрыта лишь неформально, является *неестественной* для целей программы, не очень хорошо документирована и не полностью принимается в мире Python. Как было показано, даже при хорошем понимании алгоритма MRO важно помнить о том, что он оказывает весьма тонкое влияние на *настройку*, *связность* и *гибкость*. Если вы не вполне

понимаете работу данного алгоритма или он не решает цели вашего приложения, тогда вероятно лучше не полагаться на неявно инициируемые им действия в своем коде.

Вот цитата из команды `import this`:

Если реализацию сложно объяснить, то она считается плохой идеей.

Вызов `super`, похоже, надежно подпадает под такую категорию. Большинство программистов не будут использовать загадочный инструмент, рассчитанный на редкий случай применения, каким бы искусным он ни был. Это особенно справедливо в языке написания сценариев, который позиционируется как дружественный в отношении неспециалистов. К сожалению, использование инструмента подобного рода любым программистом, так или иначе, навязывает его остальным — истинная причина, по которой он рассматривался здесь, и тема, к которой мы еще вернемся в конце книги.

Как всегда, время и пользовательская база покажут, приведут ли компромиссы или преимущества вызова `super` к его более широкому принятию. Самое меньшее, он побуждает вас знать традиционную методику обращения к суперклассам по явным именам, т.к. она по-прежнему регулярно применяется и часто либо проще, либо является обязательной в современном программировании на Python. Если вы решите использовать вызов `super`, тогда рекомендую помнить о том, что применение `super`:

- в режиме одиночного наследования с разрастанием деревьев может маскировать будущие проблемы и приводить к неожиданному поведению;
- в режиме множественного наследования привносить значительную сложность в нестандартный сценарий использования Python.

Поищите соответствующие статьи в веб-сети, чтобы ознакомиться с другими мнениями по поводу вызова `super` в Python и узнать дальнейшие детали о его хороших и плохих сторонах. Вы можете обнаружить массу дополнительных точек зрения, хотя в конечном итоге будущее Python в равной степени зависит от вас, как и от всех остальных.

Также в главе 40 предлагается формальное описание полного наследования — процедуры, которая исключает объекты `super` из специального просмотра хвоста MRO, специфичного для контекста, попутно выполняя поиск первого вхождения атрибута (дескриптора или значения). Полное наследование применяется на самом объекте `super`, только если этот просмотр потерпел неудачу. Совокупным эффектом будет особый случай для базового распознавания имен, навязываемый языком и вашим кодом ради относительно редкого сценария использования.

Затруднения, связанные с классами

Итак, мы добрались до конца основного раскрытия темы ООП в настоящей книге. Сразу после исключений мы еще будем исследовать примеры и вопросы, касающиеся классов, в последней части книги, но там просто предлагается расширенный обзор концепций, которые были введены здесь. Как обычно, текущая часть книги стандартно завершается предостережениями о ловушках, которых следует избегать.

Большинство проблем с классами можно свести к проблемам, связанным с пространствами имен. Сказанное имеет смысл, принимая во внимание тот факт, что классы представляют собой всего лишь пространства имен с несколькими дополнительными тонкостями. Некоторые пункты в данном разделе больше похожи на указания по применению классов, чем на проблемы, но, как известно, с ними сталкиваются даже опытные разработчики классов.

Изменение атрибутов классов может иметь побочные эффекты

Теоретически классы (и экземпляры классов) являются *изменяемыми* объектами. Подобно встроенным спискам и словарям вы можете изменять их на месте за счет присваивания значений их атрибутам – и как в случае списков и словарей изменение объекта класса или экземпляра может оказать влияние на множество ссылок на него.

Обычно это то, чего мы хотим, и в целом именно так объекты изменяют свое состояние, но осведомленность об указанной проблеме становится особенно важной, когда изменяются атрибуты классов. Так как все экземпляры, созданные из класса, разделяют пространство имен класса, любые изменения на уровне класса отражаются во всех экземплярах, если только они не имеют собственные версии измененных атрибутов класса.

Поскольку классы, модули и экземпляры представляют собой просто объекты с пространствами имен атрибутов, вы можете нормально изменять их атрибуты во время выполнения через присваивание. Рассмотрим показанный ниже класс X. Внутри тела класса присваивание значения имени a создает атрибут X.a, который во время выполнения находится в объекте класса и будет унаследован всеми экземплярами класса X:

```
>>> class X:
        a = 1        # Атрибут класса

>>> I = X()
>>> I.a              # Наследуется экземпляром
1
>>> X.a
1
```

Пока все хорошо – мы имеем дело с обычным случаем. Но обратите внимание, что происходит, когда мы изменяем атрибут класса динамически за пределами оператора class: в итоге также изменяется атрибут в каждом объекте, унаследованном от класса. Кроме того, новые экземпляры, созданные из класса в течение текущего сеанса или запуска программы, также получают динамически установленное значение независимо от того, что указано в исходном коде класса:

```
>>> X.a = 2          # Может изменить не только X
>>> I.a              # I тоже изменяется
2
>>> J = X()          # J наследует значения времени выполнения от X
>>> J.a              # (но присваивание значения J.a изменяет a в J, но не X либо I)
2
```

Это полезная возможность или опасная ловушка? Судить вам. Как было показано в главе 27, фактически вы можете сделать работу за счет изменения атрибутов класса даже без создания одиночного экземпляра – методика, с помощью которой удастся эмулировать использование "записей" или "структур" в других языках. В качестве напоминания взгляните на следующую необычную, но допустимую программу на Python:

```
class X: pass      # Создать несколько пространств имен атрибутов
class Y: pass

X.a = 1            # Использовать атрибуты классов как переменные
X.b = 2            # Какие-либо экземпляры отсутствуют
X.c = 3
Y.a = X.a + X.b + X.c
for X.i in range(Y.a): print(X.i)    # Выводится 0..5
```

Здесь классы X и Y работают подобно модулям "без файлов" – пространствам имен для хранения переменных, которые не должны конфликтовать. Это вполне законный трюк в программировании на Python, но он менее уместен, когда применяется к классам, реализованным другими; вы не всегда можете быть уверены в том, что изменяемые атрибуты отдельного класса не являются критически важными для его внутреннего поведения. Если вы намереваетесь эмулировать структуру из языка С, тогда может быть лучше изменять экземпляры, нежели классы, т.к. подобным способом оказывается воздействие только на один объект:

```
class Record: pass
X = Record()
X.name = 'bob'
X.job = 'Pizza maker'
```

Модификация изменяемых атрибутов классов тоже может иметь побочные эффекты

На самом деле это затруднение является расширением предыдущего. Из-за того, что атрибуты класса разделяются всеми экземплярами, если какой-то атрибут ссылается на изменяемый объект, то изменение данного объекта на месте внутри любого экземпляра повлияет сразу на все экземпляры:

```
>>> class C:
        shared = []                # Атрибуты класса
        def __init__(self):
           self.perobj = []        # Атрибуты экземпляра

>>> x = C()                        # Два экземпляра
>>> y = C()                        # Неявно разделяет атрибуты класса
>>> y.shared, y.perobj
([], [])

>>> x.shared.append('spam')        # Влияет также на представление y!
>>> x.perobj.append('spam')        # Влияет только на данные x
>>> x.shared, x.perobj
(['spam'], ['spam'])

>>> y.shared, y.perobj             # y видит изменения, внесенные через x
(['spam'], [])
>>> C.shared                       # Хранятся в классе и разделяются
['spam']
```

Результат ничем не отличается от того, что мы неоднократно наблюдали в книге: изменяемые объекты разделяются простыми переменными, глобальные имена – функциями, объекты уровня модулей – множеством импортеров, а изменяемые аргументы функций – вызывающим и вызываемым кодом. Все они представляют собой сценарии общепринятого поведения (много ссылок на изменяемый объект) и все подвержены влиянию, если разделяемый объект изменяется на месте по любой ссылке. Здесь подобное происходит в атрибутах класса, разделяемых всеми экземплярами через наследование, но мы имеем дело с аналогичным явлением. Его можно сделать более тонким за счет разного поведения присваивания значений самим атрибутам экземпляра:

```
x.shared.append('spam')     # Изменяет на месте разделяемый объект,
                            #   присоединенный к классу
x.shared = 'spam'           # Изменяет или создает атрибут экземпляра,
                            #   присоединенный к x
```

Но снова это не проблема, а просто то, о чем нужно знать; с разделяемыми изменяемыми атрибутами классов связаны многие допустимые сценарии использования в программах на Python.

Множественное наследование: порядок имеет значение

Сейчас это может быть очевидным, но полезно подчеркнуть: если вы применяете множественное наследование, тогда порядок перечисления суперклассов в заголовке оператора `class` может быть важным. Интерпретатор Python всегда осуществляет поиск в суперклассах слева направо согласно порядку их следования в строке заголовка.

Предположим, что в примере с множественным наследованием из главы 31 класс `Super` тоже реализует метод `__str__`:

```
class ListTree:
    def __str__(self): ...

class Super:
    def __str__(self): ...

class Sub(ListTree, Super):    # Получить метод __str__ из ListTree
                               #   за счет указания его первым

x = Sub()                      # При наследовании поиск осуществляется
                               #   в ListTree перед Super
```

От какого класса мы унаследуем метод `__str__` – `ListTree` или `Super`? Так как поиск при наследовании выполняется слева направо, мы получим метод из класса, находящегося в списке первым (крайним слева) в заголовке оператора `class` для `Sub`. По-видимому, мы указали бы класс `ListTree` первым, потому что все его предназначение связано со специальным методом `__str__` (в действительности мы должны были сделать это в главе 31 при смешивании данного класса с `tkinter.Button`, который и сам имеет метод `__str__`).

Но теперь предположим, что `Super` и `ListTree` имеют собственные версии атрибутов с такими же именами. Если нам необходимо одно имя из `Super` и еще одно из `ListTree`, то порядок их перечисления в заголовке оператора `class` не поможет – мы должны будем переопределить наследование, вручную выполнив присваивание значения имени атрибута в классе `Sub`:

```
class ListTree:
    def __str__(self): ...
    def other(self): ...

class Super:
    def __str__(self): ...
    def other(self): ...

class Sub(ListTree, Super):    # Получить метод __str__ из класса ListTree,
                               #   указав его первым
    other = Super.other        # Но явно выбрать версию other из класса Super
    def __init__(self):
        ...

x = Sub()                      # Поиск при наследовании производится в Sub
                               #   и затем в ListTree/Super
```

Присваивание значения `other` внутри класса `Sub` создает `Sub.other` – ссылку на объект `Super.other`. Поскольку она находится ниже в дереве, `Sub.other` фактически скрывает `ListTree.other`, т.е. атрибут, который обычно обнаруживается при поис-

ке процедурой наследования. Аналогично, если мы укажем Super первым в заголовке оператора class, чтобы выбрать ее атрибут other, тогда метод ListTree придется выбирать явно:

```
class Sub(Super, ListTree):   # Получить other из Super согласно порядка
    __str__ = Lister.__str__  # Явно выбрать Lister.__str__
```

Множественное наследование — сложный инструмент. Даже в случае понимания последнего абзаца все равно лучше использовать его умеренно и осмотрительно. Иначе смысл имени может стать зависимым от порядка, в котором классы смешиваются в произвольно отдаленном подклассе. (Еще один пример применения методики, продемонстрированной здесь в действии, был приведен при обсуждении явного устранения конфликтов в разделе "Модель классов 'нового стиля'" ранее в главе, а также вызова super.)

В качестве эмпирического правила запомните, что множественное наследование работает лучше, когда ваши подмешиваемые классы являются настолько независимыми, насколько возможно — так как они могут использоваться в разнообразных контекстах, в них не должны выдвигаться допущения об именах, относящихся к другим классам в дереве. Средство псевдозакрытых атрибутов __*X*, рассмотренное в главе 31, способно помочь за счет локализации имен, на которые полагается сам класс, и ограничивает имена, добавляемые подмешиваемыми классами. В приведенном примере, если класс ListTree предназначен только для экспортирования своего специального метода __str__, то он может назначить своему методу other имя __other, чтобы избежать конфликта с идентично именованными методами в классах внутри дерева.

Области видимости в методах и классах

При продумывании смысла имен в коде, основанном на классах, полезно помнить о том, что классы в точности как функции вводят локальные области видимости, а методы являются просто более вложенными функциями. В следующем примере функция generate возвращает экземпляр вложенного класса Spam. Внутри своего кода имя класса Spam присваивается в локальной области видимости функции generate и потому оно доступно любой вложенной функции, включая код в method; это соответствует букве *E* в правиле LEGB:

```
def generate():
    class Spam:             # Spam - имя в локальной области видимости generate
        count = 1
        def method(self):
            print(Spam.count)   # Согласно правилу LEGB (E) доступно
                                #  в области видимости generate
        return Spam()

generate().method()
```

Пример работает, начиная с версии Python 2.2, потому что локальные области видимости всех операторов def объемлющих функций автоматически видны вложенным def (в том числе вложенные def для методов, как было в примере).

Даже при этих условиях имейте в виду, что операторы def для методов не могут видеть локальную область видимости включающего *класса*; они способны видеть только локальные области видимости объемлющих def. Вот почему методы обязаны указывать экземпляр self либо имя класса, чтобы ссылаться на методы и другие атрибуты, определенные во включающем операторе class. Скажем, в коде метода должна применяться ссылка self.count или Spam.count, но не просто count.

Во избежание вложения мы могли бы реструктурировать код, так чтобы класс Spam определялся на верхнем уровне модуля. Тогда вложенная функция method и функция верхнего уровня generate будут находить класс Spam в своих глобальных областях видимости; он не локализуется внутри области видимости какой-то функции, но по-прежнему является локальным по отношению к одиночному модулю:

```
def generate():
    return Spam()

class Spam:                    # Определение на верхнем уровне модуля
    count = 1
    def method(self):
        print(Spam.count)      # Работает: в глобальной области видимости
                               #  (включающий модуль)

generate().method()
```

На самом деле такой подход рекомендуется для всех выпусков Python — если избегать вложения классов и функций, то код становится в целом проще. С другой стороны, вложение классов полезно в контекстах *замыканий*, где область видимости объемлющей функции предохраняет *состояние*, используемое классом или его методами. В приведенном далее взаимодействии вложенная функция method имеет доступ к собственной области видимости, области видимости объемлющей функции (для label), глобальной области видимости включающего модуля, всему сохраненному в экземпляре self классом и самому классу через его имя nonlocal:

```
>>> def generate(label):       # Возвращает класс вместо экземпляра
        class Spam:
            count = 1
            def method(self):
                print("%s=%s" % (label, Spam.count))
        return Spam

>>> aclass = generate('Gotchas')
>>> I = aclass()
>>> I.method()
Gotchas=1
```

Другие затруднения, связанные с классами

В качестве обзора ниже предлагается несколько дополнительных предостережений, касающихся классов.

Разумно выбирайте хранилище для каждого экземпляра или класса

Аналогичным образом будьте осмотрительны, когда решаете, должен ли конкретный атрибут храниться в классе или в его экземплярах: в первом случае атрибут разделяется всеми экземплярами, а во втором он будет отличаться в каждом экземпляре. На практике это может быть критически важным вопросом проектирования. Например, если в программе с графическим пользовательским интерфейсом нужно, чтобы информация разделялась всеми объектами оконного класса, которые будет создавать приложение (скажем, каталог, куда операция сохранения выполняла запись в последний раз, или уже введенный пароль), тогда она должна храниться как данные уровня класса; при хранении в экземпляре как атрибутов self информация будет варьироваться в каждом окне или теряться во время поиска процедурой наследования.

Вам обычно понадобится вызывать конструкторы суперклассов

Помните о том, что при создании экземпляра интерпретатор Python выполняет только один метод конструктора `__init__` — самый нижний в дереве наследования классов. Он не запускает автоматически конструкторы всех суперклассов, расположенных выше в дереве. Поскольку конструкторы, как правило, выполняют обязательную работу по начальному запуску, вам обычно необходимо будет запускать конструктор суперкласса из конструктора подкласса. Для этого вы будете применять ручной вызов через имя суперкласса (или `super`) и передавать ему любые требующиеся аргументы, если только в ваши намерения не входит полное замещение конструктора суперкласса или же суперкласс вообще не имеет и не наследует какой-нибудь конструктор.

Классы на основе делегирования в Python 3.X: `__getattr__` и встроенные операции

Еще одно напоминание: как было описано ранее в главе и в других местах, классы, которые используют метод перегрузки операции `__getattr__` для делегирования операций извлечения атрибутов внутренним объектам, могут потерпеть неудачу в Python 3.X (и в Python 2.X в случае применения классов нового стиля), если не были переопределены методы перегрузки операций в классе оболочки. Имена методов перегрузки операций, неявно извлекаемые встроенными операциями, не направляются обобщенным методам перехвата атрибутов. Чтобы обойти проблему, вам придется переопределять такие методы в классах оболочек либо вручную посредством инструментов, либо за счет определения в суперклассах; в главе 40 мы покажем, как это делать.

Еще раз о KISS: чрезмерно большое количество уровней

При надлежащей эксплуатации возможности ООП по многократному использованию кода позволяют значительно сократить время разработки. Однако иногда потенциал абстракции ООП применяется до такой степени неправильно, что делает код трудным для понимания. Если иерархия классов оказывается слишком глубокой, тогда код может стать непонятным; для выяснения, что делает та или иная операция, вам придется просматривать множество классов.

Например, когда-то я имел дело с системой на C++, насчитывающей тысячи классов (некоторые являлись сгенерированными машиной) и имеющей до 15 уровней наследования. Расшифровка вызовов методов в настолько сложной системе часто была внушительной задачей: даже для самых элементарных операций приходилось обращаться за справкой к многочисленным классам. Фактически логика системы была разнесена на такое большое количество уровней, что понимание фрагмента кода в ряде случаев требовало многодневного копания в связанных файлах. Для продуктивности программиста это очевидно неидеально!

Здесь тоже применимо наиболее общее правило программирования на Python: *не делайте вещи сложными, если только они по-настоящему не должны быть такими.* Разнесение кода по множеству уровней классов, вплоть до непонятности, всегда считается плохой идеей. Абстракция является основой полиморфизма и инкапсуляции и при корректном использовании может быть крайне эффективным инструментом. Тем не менее, вы упростите отладку и удобство сопровождения, если сделаете интерфейсы своих классов интуитивно понятными, откажетесь от излишне абстрактного кода и постараетесь сохранять иерархии классов неглубокими, разве только не возникнет веская причина поступать иначе. Запомните: код, который вы пишете, обычно будет тем кодом, который придется читать другим. За обсуждением KISS обращайтесь в главу 20 первого тома.

Резюме

В главе был представлен набор расширенных тем, связанных с классами, в том числе создание подклассов из встроенных типов, классы нового стиля, статические методы и декораторы. Большинство из них являются необязательными расширениями модели ООП на Python, но они станут более полезными, когда вы приступите к реализации крупных объектно-ориентированных программ, а их знание пригодится, если вам придется разбирать код, в которых они встречаются. Как упоминалось ранее, обсуждение ряда сложных инструментов для работы с классами продолжится в финальной части книги; там будут даны дополнительные сведения о свойствах, дескрипторах, декораторах и метаклассах.

Мы подошли к окончанию части, посвященной классам, и потому в конце главы вы найдете традиционные учебные упражнения: постарайтесь проработать их, чтобы попрактиковаться в написании кода реальных классов. В следующей главе мы начнем рассматривать последнюю основную тему языка, *исключения* – механизм Python для сообщения своему коду об ошибках и других условиях. Тема относительно легковесна, но я приберег ее напоследок, потому что новые исключения в наши дни должны быть реализованы в виде классов. Однако прежде чем двигаться дальше, ответьте на контрольные вопросы главы и выполните предложенные упражнения.

Проверьте свои знания: контрольные вопросы

1. Назовите два способа расширения встроенного объектного типа.
2. Для чего используются декораторы функций и классов?
3. Как реализовать класс нового стиля?
4. Чем отличаются классы нового стиля от классических классов?
5. Чем отличаются нормальные методы от статических методов?
6. Допустимо ли применять в коде инструменты вроде `__slots__` и `super`?
7. Сколько нужно выждать, прежде чем бросать "Пресвятую Антиохийскую Гранату"?

Проверьте свои знания: ответы

1. Вы можете внедрить встроенный объект в класс оболочки или создать подкласс из встроенного типа напрямую. Последний подход имеет тенденцию быть проще, т.к. автоматически наследуется большинство исходных линий поведения.
2. Декораторы функций обычно используются для управления функцией или методом либо для добавления уровня логики, которая выполняется каждый раз, когда функция или метод вызывается. Они могут применяться для регистрации в журнале или подсчета вызовов функции, проверки ее типов аргументов и т.д. Они также используются для "объявления" статических методов (простых функций в классе, которым при вызове не передается экземпляр), а также методов и свойств класса. Декораторы классов похожи, но вместо обращений к функциям управляют целыми объектами и их интерфейсами.

3. Классы нового стиля реализуются путем наследования от встроенного класса `object` (или любого другого встроенного типа). В Python 3.X все классы автоматически являются классами нового стиля, поэтому такое наследование не требуется (но ничем не повредит); в Python 2.X классы, явно унаследованные от `object` (или любого другого встроенного типа), относятся к новому стилю, а без такого наследования будут "классическими".

4. Классы нового стиля выполняют поиск в ромбовидной схеме деревьев множественного наследования по-другому – по существу они ищут сначала в ширину (поперек), а не сначала в глубину (вверх). Классы нового стиля также изменяют результат встроенной функции `type` для экземпляров и классов, не запускают обобщенные методы извлечения атрибутов, такие как `__getattr__`, для методов встроенных операций и поддерживают набор более сложных добавочных инструментов, в числе которых свойства, дескрипторы, `super` и списки атрибутов экземпляров `__slots__`.

5. Нормальные методы (экземпляров) получают аргумент `self` (подразумеваемый экземпляр), но статические методы – нет. Статические методы являются простыми функциями, вложенными в объекты классов. Чтобы сделать метод статическим, он должен либо запускаться через специальную встроенную функцию, либо быть декорирован с помощью синтаксиса декораторов. В Python 3.X простые функции в классе разрешено вызывать через класс без указанного шага, но вызовы через экземпляры по-прежнему требуют объявления статических методов.

6. Конечно, но вы не должны автоматически использовать расширенные инструменты без тщательного анализа последствий. Скажем, слоты способны нарушить работу кода; вызов `super` может маскировать будущие проблемы, когда применяется для одиночного наследования, а при множественном наследовании привносит значительную сложность для изолированного сценария использования; и для максимальной пользы оба инструмента требуют повсеместного применения. Оценка новых или продвинутых инструментов является главной задачей любого инженера, вот почему мы настолько внимательно исследовали связанные с ними компромиссы. Цель этой книги не в том, чтобы сообщить вам, какие инструменты использовать, а подчеркнуть важность их объективного анализа – задачи, которой зачастую назначают слишком низкий приоритет в области разработки программного обеспечения.

7. Три секунды. (Вот более точная цитата: "... и рек Господь: 'Допреже всего Пресвятую Чеку извлечь долженствует. Опосля же того сочти до трех, не более и не менее. Три есть цифирь, до коей счесть потребно, и сочтенья твои суть три. До четырех счесть не моги, паче же до двух, опричь токмо коли два предшествует трем. О пяти и речи быть не может. Аще же достигнешь ты цифири три, что есть и пребудет третьею цифирью, брось Пресвятою Антиохийскою Гранатою твоею во врага твоего, и оный враг, за все проказы пред лицом моим, окочурится.'")[2]

[2] `https://ru.wikiquote.org/wiki/Монти_Пайтон_и_Священный_Грааль`

Проверьте свои знания: упражнения для части VI

В приведенных ниже упражнениях вам предлагается реализовать несколько классов и поэкспериментировать с некоторым существующим кодом. Несомненно, проблема с существующим кодом в том, что он должен существовать. Чтобы поработать с классом множества в упражнении 5, либо загрузите исходный код для книги из веб-сайта издательства, либо наберите его вручную (он довольно короткий). Программы начинают становиться более сложными, поэтому обязательно просмотрите решения упражнений, находящиеся в приложении Г.

1. ***Наследование.*** Напишите класс по имени `Adder`, который экспортирует метод `add(self, x, y)`, выводящий сообщение `Not Implemented` (Не реализован). Затем определите два подкласса класса `Adder`, которые реализуют метод `add`:

   `ListAdder`

   С методом `add`, который возвращает сцепление своим двух списковых аргументов.

   `DictAdder`

   С методом `add`, который возвращает новый словарь, содержащий элементы из его двух словарных аргументов (подойдет любое определение словарного дополнения).

   Поэкспериментируйте с интерактивным созданием экземпляров всех трех классов и вызовом их методов `add`. Затем расширьте суперкласс `Adder` для сохранения объекта в экземпляре с помощью конструктора (например, присвоив `self.data` список или словарь) и перегрузите операцию + посредством метода `__add__`, чтобы автоматически направлять его вашим методам `add` (скажем, `X + Y` запускает `X.add(X.data,Y)`). Где лучше всего разместить конструкторы и методы перегрузки операций (т.е. в каких классах)? Какие разновидности объектов вы можете добавлять в экземпляры своих классов?

   На практике вы можете обнаружить, что методы `add` легче реализовать для приема только одного реального аргумента (например, `add(self,y)`) и добавлять этот один аргумент к текущим данным экземпляра (наподобие `self.data + y`). Имеет ли больше смысла поступать так, чем передавать `add` два аргумента? Сказали бы вы, что в итоге классы стали более "объектно-ориентированными"?

2. ***Перегрузка операций.*** Напишите класс по имени `MyList`, который скрывает ("помещает внутрь себя") список Python: он должен перегружать большинство списковых операций, включая +, индексирование, итерацию, нарезание, а также списковые методы, такие как `append` и `sort`. Перечень всех возможных методов для поддержки ищите в справочном руководстве по Python или в другой документации. К тому же предоставьте для своего класса конструктор, который принимает существующий список (или экземпляр `MyList`) и копирует его компоненты в атрибут экземпляра. Поэкспериментируйте с классом интерактивно. Вот что вам нужно будет выяснить.

   а) Почему здесь важно копирование начального значения?

   б) Можете ли вы использовать пустой срез (например, `start[:]`) для копирования начального значения, если он является экземпляром `MyList`?

в) Имеется ли общий способ для маршрутизации внутреннему списку обращений к списковым методам?

г) Можете ли вы сложить экземпляр MyList и нормальный список? Как насчет списка и экземпляра MyList?

д) Какой тип должны возвращать операции вроде + и нарезания? Как насчет операций индексирования?

е) Если вы работаете с относительно свежим выпуском Python, то можете реализовать класс оболочки такого рода за счет внедрения реального списка внутрь автономного класса или путем расширения встроенного спискового типа посредством подкласса. Что легче и почему?

3. Создание подклассов. Создайте подкласс по имени MyListSub для класса MyList из упражнения 2, который расширяет MyList с целью вывода сообщения в stdout перед каждым обращением к перегруженной операции + и подсчета количества таких обращений. Подкласс MyListSub должен наследовать базовое поведение методов от MyList. Добавление последовательности к MyListSub должно инициировать вывод сообщения, инкрементирование счетчика обращений к операции + и выполнение метода из суперкласса. Кроме того, введите новый метод, который выводит счетчики операций в stdout, и поэкспериментируйте с классом в интерактивном сеансе. Подсчитывают ли ваши счетчики обращения по экземплярам или по классу (для всех экземпляров класса)? Как бы вы реализовали другой вариант? (Подсказка: это зависит от того, какому объекту назначены члены счетчиков — члены класса разделяются экземплярами, но члены self являются данными для каждого экземпляра.)

4. Методы атрибутов. Напишите класс по имени Attrs с методами, которые перехватывают каждый случай указания атрибутов (извлечение и присваивание) и выводят в stdout сообщения со списком их аргументов. Создайте экземпляр Attrs и поэкспериментируйте интерактивно с его уточнением. Что происходит, когда вы пытаетесь использовать экземпляр в выражениях? Попробуйте добавлять, индексировать и нарезать экземпляр своего класса. (Примечание: полностью обобщенный подход, основанный на __getattr__, будет работать в классических классах Python 2.X, но не в классах нового стиля Python 3.X, которые необязательны в Python 2.X, по причинам, отмеченным в главах 28, 31 и 32 и подытоженным в решении этого упражнения.)

5. Объекты множества. Поэкспериментируйте с классом множества, описанным в разделе “Расширение типов путем внедрения” настоящей главы. Запустите команды для выполнения перечисленных ниже операций.

а) Создайте два множества целых чисел и получите их пересечение и объединение с применением операций выражений & и |.

б) Создайте множество из строки и поэкспериментируйте с его индексированием. Какие методы класса вызываются?

в) Попробуйте пройти по элементам внутри строкового множества, используя цикл for. Какие методы запускаются на этот раз?

г) Попробуйте получить пересечение и объединение строкового множества и простой строки Python. Работает ли это?

д) Расширьте свое множество, создав подкласс для обработки произвольно большого количества операндов с применением формы аргументов *args.

(Подсказка: просмотрите функциональные версии этих алгоритмов в главе 18 первого тома.) Вычислите пересечения и объединения множества операндов с помощью нового подкласса множества. Как можно получить пересечение трех и более множеств, учитывая наличие у операции & только двух сторон?

e) Как бы вы обошлись с эмуляцией других списковых операций в классе множества? (Подсказка: `__add__` может перехватывать конкатенацию, а `__getattr__` – передавать большинство вызовов именованных списковых методов вроде `append` внутреннему списку.)

6. ***Связи деревьев классов.*** В разделе "Словари пространств имен: обзор" главы 29 и в разделе "Множественное наследование: 'подмешиваемые' классы" главы 31 вы узнали, что классы имеют атрибут `__bases__`, который возвращает кортеж их объектов суперклассов (тех, что перечисляются внутри круглых скобок в заголовке оператора `class`). Используйте `__bases__` для расширения подмешиваемых классов `lister.py` из главы 31, чтобы они выводили имена непосредственных суперклассов класса, к которому относится экземпляр. Когда вы завершите, первая строка в строковом представлении должна выглядеть примерно так (адрес у вас наверняка будет другим):

```
<Instance of Sub(Super, Lister), address 7841200:
```

7. ***Композиция.*** Смоделируйте сценарий заказа блюд быстрого питания, определив четыре класса:

`Lunch`

Класс контейнера и контроллера

`Customer`

Агент, который покупает блюдо

`Employee`

Агент, у которого заказчик производит заказ

`Food`

То, что заказчик покупает

Для начала вот классы и методы, которые вы определите:

```
class Lunch:
    def __init__(self)    # Создает/внедряет экземпляры Customer и Employee
    def order(self, foodName) # Начинает эмуляцию заказа экземпляром Customer
    def result(self) # Запрашивает у Customer, какой экземпляр Food он имеет

class Customer:
    def __init__(self)            # Инициализирует блюдо значением None
    def placeOrder(self, foodName, employee)    # Размещает заказ с
                                                #   экземпляром Employee
    def printFood(self)                         # Выводит название блюда

class Employee:
    def takeOrder(self, foodName)               # Возвращает экземпляр Food
                                                #   с запрошенным названием

class Food:
    def __init__(self, name)                    # Сохраняет название блюда
```

Эмуляция заказа должна работать так, как описано ниже.

а) Конструктор класса Lunch должен создавать и внедрять экземпляр Customer и экземпляр Employee, а также экспортировать метод по имени order. Метод order должен запрашивать у Customer размещение заказа, вызывая его метод placeOrder. В свою очередь метод placeOrder класса Customer должен запрашивать у объекта Employee новый объект Food, вызывая метод takeOrder класса Employee.

б) Объект Food должен хранить строку с названием блюда, переданную из Lunch.order методу Customer.placeOrder, методу Employee.takeOrder и, наконец, конструктору Food. Класс верхнего уровня Lunch также должен экспортировать метод по имени result, который запрашивает у заказчика вывод названия блюда, полученное им от Employee через заказ (это можно применять для тестирования эмуляции).

Обратите внимание, что экземпляру Lunch необходимо передать либо экземпляр Employee, либо самого себя экземпляру Customer, чтобы предоставить возможность Customer вызывать методы Employee.

Поэкспериментируйте со своими классами в интерактивном сеансе путем импортирования класса Lunch, вызова его метода order для запуска взаимодействия и затем вызова его метода result для проверки, что Customer получил то, что было заказано. При желании можете также реализовать тестовые сценарии в виде кода самотестирования в файле, где определены классы, используя прием с атрибутом __name__ модуля из главы 25 первого тома. В этой эмуляции Customer является активным агентом; как бы вы изменили свои классы, если бы взамен Employee был объектом, который инициирует взаимодействие заказчик/сотрудник?

8. Иерархия для представления животных в зоопарке. Взгляните на дерево классов, показанное на рис. 32.1.

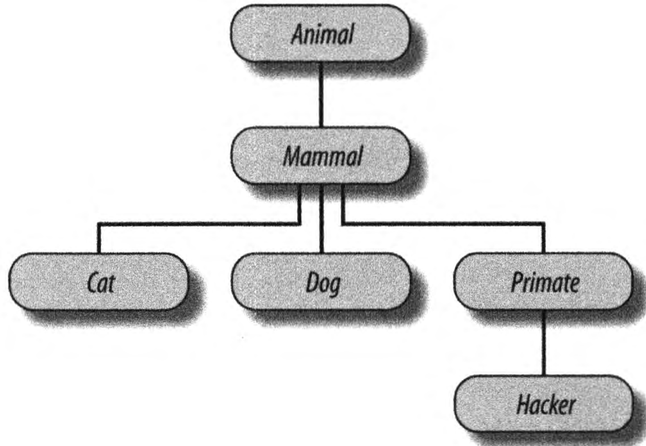


Рис. 32.1. Иерархия для представления животных в зоопарке, образованная из классов, которые связаны в дерево для поиска атрибутов при наследовании. Классы животных располагают общим методом reply*, но каждый класс может иметь собственный специальный метод* speak*, вызываемый методом* reply

Напишите набор из шести операторов `class` для моделирования такой иерархической классификации с помощью *наследования* Python. Затем добавьте в каждый класс метод `speak`, выводящий уникальное сообщение, а в суперкласс верхнего уровня `Animal` – метод `reply`, который просто вызывает `self.speak`, чтобы запустить инструмент вывода, специфичный для категории, из подкласса ниже в дереве (это инициирует независимый поиск при наследовании из `self`). Наконец, удалите метод `speak` из класса `Hacker`, чтобы он выбирал стандартный метод, находящийся выше. Когда вы завершите, ваши классы должны работать следующим образом:

```
% python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
>>> spot.reply()          # Animal.reply: вызывает Cat.speak
meow
>>> data = Hacker()       # Animal.reply: вызывает Primate.speak
>>> data.reply()
Hello world!
```

9. ***Скетч "Мертвый попугай"***. Взгляните на структуру внедрения объектов, изображенную на рис. 32.2.

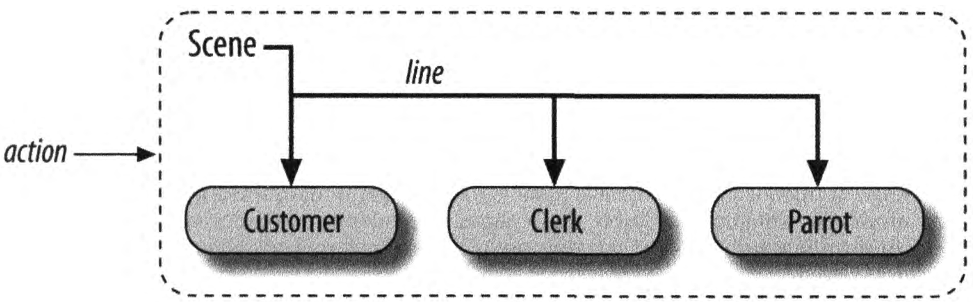


***Рис. 32.2.** Композиция сцены с классом контроллера (`Scene`), который внедряет и управляет экземплярами остальных трех классов (`Customer`, `Clerk`, `Parrot`). Классы внедренных экземпляров также могут принимать участие в иерархии наследования; композиция и наследование часто являются одинаково удобными способами структуризации классов для поддержки многократного использования кода*

Напишите набор классов Python для реализации этой структуры с помощью *композиции*. В объекте `Scene` должен быть определен метод `action` и внедрены экземпляры классов `Customer`, `Clerk` и `Parrot` (каждый из которых должен определять метод `line`, выводящий уникальное сообщение). Внедренные объекты могут либо наследовать метод `line` от общего суперкласса и просто предоставлять текст сообщения, либо определять `line` самостоятельно. В конце ваши классы должны оперировать примерно так:

```
% python
>>> import parrot
>>> parrot.Scene().action()        # Активизирует вложенные объекты
customer: "that's one ex-bird!"
clerk: "no it isn't..."
parrot: None
```

Что потребует внимания: ООП глазами экспертов

Когда я веду учебные курсы по Python, то неизменно обнаруживаю, что приблизительно на середине курса люди, которые занимались ООП в прошлом, внимательно слушают, в то время как люди, не имеющие такого опыта, сидят с осоловевшими глазами (или вообще начинают клевать носом). Смысл технологии попросту неочевиден.

В книге подобного рода я могу себе позволить роскошь включать материалы вроде обзора общей картины в главе 26 и пошагового обучающего руководства в главе 28 – на самом деле, возможно, вам следует пересмотреть эти главы, если вы склонны считать, что ООП является чем-то бессмысленным. Хотя ООП привносит гораздо больше структуры, чем рассмотренные ранее генераторы, оно похожим образом опирается на некую магию (поиск при наследовании и особый первый аргумент), дать рациональное объяснение которой начинающим может быть затруднительно.

Тем не менее, в реальных группах, чтобы помочь новичкам уловить суть (и не дать им уснуть), я обычно останавливаюсь и спрашиваю экспертов в аудитории, почему они применяют ООП. Их ответы могут пролить свет на замысел ООП, если эта темя для вас нова.

Ниже приведен слегка приукрашенный перечень самых распространенных причин использования ООП, на которые ссылались мои студенты на протяжении многих лет.

Многократное использование кода

Это самая легкая для понимания и главная причина применения ООП. За счет поддержки наследования классы позволяют вам программировать путем настройки, а не начинать каждый проект с нуля.

Инкапсуляция

Помещение деталей реализации позади объектных интерфейсов изолирует пользователей класса от изменений в коде.

Структура

Классы предоставляют новые локальные области видимости, которые сводят к минимуму конфликты имен. Они также обеспечивают естественное место для написания и поиска кода реализации и для управления состоянием объектов.

Сопровождение

Классы естественным образом способствуют разложению кода на элементарные операции, что позволяет минимизировать избыточность. Благодаря структуре и поддержке многократного использования кода обычно приходится изменять только одну копию кода.

Согласованность

Классы и наследование позволяют вам реализовывать общие интерфейсы и поэтому обеспечивают общий вид и поведение в коде; в результате облегчается отладка, понимание и сопровождение.

Полиморфизм

Это скорее особенность ООП, чем причина его использования, но за счет поддержки универсальности кода полиморфизм делает код более гибким и широко применимым, а потому более пригодным для многократного использования.

Другие

И, конечно же, причина номер один для применения ООП, которую называли студенты: его упоминание оно здорово смотрится в резюме! (Ладно, я привел это как шутку, но важно быть знакомым с ООП, если вы планируете работать в современной индустрии программного обеспечения.)

Наконец, помните о сказанном мною в начале данной части книги: вы не сможете по достоинству оценить ООП, пока не позанимаетесь им какое-то время. Выберите проект, изучите более крупные примеры, проработайте упражнения – все это заставит вас попотеть над объектно-ориентированным кодом, но приложенные усилия того стоят.

ЧАСТЬ VII

Исключения и инструменты

ГЛАВА 33

Основы исключений

В этой части книги мы будем иметь дело с *исключениями*, которые являются событиями, способными изменить поток управления в программе. Исключения в Python возникают автоматически при ошибках и могут генерироваться и перехватываться вашим кодом. Они обрабатываются четырьмя операторами, рассматриваемыми в данной части, первый из которых имеет две вариации (перечисленные ниже по отдельности), а последний был необязательным расширением вплоть до версий Python 2.6 и Python 3.0.

try/except

Перехватывает и производит восстановление после исключений, инициируемых Python или вами.

try/finally

Выполняет действия по очистке независимо от того, происходили исключения или нет.

raise

Генерирует исключение вручную в коде.

assert

Генерирует исключение условно в коде.

with/as

Реализует диспетчеры контекстов в Python 2.6, 3.0 и последующих версиях (необязательные в Python 2.5).

Рассмотрение этой темы перенесено ближе к концу книги, т.к. для реализации самих исключений вам необходимо было изучить классы. Однако за небольшими исключениями (получился каламбур) вы обнаружите, что обработка исключений в коде на Python проста из-за ее интеграции в сам язык как еще одного высокоуровневого инструмента.

Для чего используются исключения?

Вкратце исключения позволяют нам перескакивать через произвольно большие порции кода программы. Возьмем обсуждаемый ранее в книге гипотетический робот по приготовлению пиццы. Предположим, что мы занялись идеей всерьез и построили

такую машину. Чтобы приготовить пиццу, нашему кулинарному автомату понадобится выполнить план, который мы реализуем в виде программы на Python: она примет заказ, замесит тесто, добавит начинки, испечет основу и т.д.

Теперь представим, что во время шага "испечет основу" что-то пошло совершенно не так. Возможно, сломался духовой шкаф или робот неправильно рассчитал расстояние для перемещения к нему и самопроизвольно воспламенился. Очевидно, мы хотим иметь возможность перехода на код, который быстро обработает такие состояния. Поскольку в необычных случаях подобного рода у нас нет никакой надежды на то, что задача приготовления пиццы будет доведена до конца, мы также могли бы целиком отказаться от плана.

Именно это и позволяют нам делать исключения: можно за один шаг перейти к обработчику исключений, отменяя все вызовы функций, которые начались до того, как был совершен вход в данный обработчик. Затем код в обработчике исключений надлежащим образом отреагирует на сгенерированное исключение (скажем, позвонив в противопожарную службу!).

Об исключении можно думать как о своеобразном структурированном "безусловном переходе". *Обработчик исключений* (оператор `try`) оставляет маркер и выполняет некоторый код. Где-то намного дальше в программе генерируется исключение, заставляющее интерпретатор Python перейти обратно на этот маркер и прекратить выполнение любых активных функций, которые были вызваны после оставления маркера. Такой протокол обеспечивает согласованный способ реагирования на необычные события. Кроме того, поскольку интерпретатор Python переходит к оператору обработчика незамедлительно, ваш код становится проще — как правило, исчезает необходимость проверять коды состояния после каждого вызова функции, которая способна потерпеть неудачу.

Роли, исполняемые исключениями

В программах на Python исключения обычно применяются для разнообразных целей. Ниже перечислены самые распространенные роли, которые они исполняют.

Обработка ошибок

Интерпретатор Python генерирует исключения всякий раз, когда обнаруживает ошибки в программах во время выполнения. Вы можете перехватывать и реагировать на ошибки в своем коде либо игнорировать инициированные исключения. Если ошибка игнорируется, тогда активизируется стандартная линия поведения обработки исключений Python: она останавливает программу и выводит сообщение об ошибке. Если вас не устраивает такое стандартное поведение, то нужно предусмотреть оператор `try` для перехвата и восстановления после исключения — при обнаружении ошибки интерпретатор Python будет переходить на ваш обработчик `try` и программа возобновит выполнение после `try`.

Уведомление о событиях

Исключения можно также использовать для оповещения о допустимых условиях, не заставляя вас передавать результирующие флаги внутри программы или явно их проверять. Например, процедура поиска могла бы генерировать исключение в случае неудачи, а не возвращать целочисленный результирующий код — и надеяться на то, что код никогда не окажется допустимым результатом!

Обработка особых случаев

Иногда условие может возникать настолько редко, что оправдать запутанность кода для его обработки в многочисленных местах довольно-таки трудно. Вы часто можете устранить код для особых случаев за счет обработки необычных ситуаций в обработчиках исключений на более высоких уровнях программы. Оператор `assert` может аналогично применяться для проверки того, что условия соответствуют ожидаемым на стадии разработки.

Действия при завершении

Как будет показано, оператор `try/finally` дает вам возможность гарантировать, что обязательные операции времени закрытия будут выполнены независимо от наличия или отсутствия исключений в ваших программах. Более новый оператор `with` предлагает в этом отношении альтернативу для объектов, которые его поддерживают.

Редкие потоки управления

Наконец, поскольку исключения являются разновидностью высокоуровневого и структурированного "безусловного перехода", вы можете их использовать в качестве основы для реализации экзотических потоков управления. Скажем, хотя в языке явно не поддерживается возврат к предыдущему состоянию, вы можете реализовать его на Python с применением исключений и небольшого объема вспомогательной логики для раскручивания присваиваний[1]. В Python не существует оператора "безусловного перехода" (к счастью!), но исключения временами способны исполнять похожие роли; например, `raise` может использоваться для выхода из множества циклов.

Некоторые из таких ролей мы кратко рассматривали ранее, и будем исследовать типичные сценарии применения исключений позже в этой части книги. А пока давайте начнем с того, что взглянем на инструменты обработки исключений Python.

Исключения: краткая история

По сравнению с рядом других тем, которые встречаются в книге, исключения представляют собой довольно легковесный инструмент в Python. Из-за их простоты мы перейдем прямо к написанию кода.

Стандартный обработчик исключений

Допустим, мы написали следующую функцию:

```
>>> def fetcher(obj, index):
        return obj[index]
```

---

[1] Однако подлинный возврат к предыдущему состоянию не является частью языка Python. Возврат к предыдущему состоянию перед переходом отменяет все вычисления, но исключения Python этого не делают: переменные, которым присваивались значения между моментом входа в оператор `try` и моментом генерации исключения, не переустанавливаются в свои предыдущие значения. Даже генераторные функции и выражения, обсуждаемые в главе 20 первого тома, не делают полный возврат к предыдущему состоянию — они реагируют на запросы `next(G)` просто восстановлением состояния и возобновлением выполнения. Дополнительные сведения о возврате к предыдущему состоянию ищите в книгах, посвященных искусственному интеллекту или языкам программирования Prolog либо Icon.

В этой функции нет ничего особенного — она всего лишь индексирует объект с использованием переданного индекса. При нормальной работе она возвращает результат по допустимому индексу:

```
>>> x = 'spam'
>>> fetcher(x, 3)              # Подобно x[3]
'm'
```

Тем не менее, если мы запросим у функции `fetcher` индексирование за концом строки, тогда возникнет исключение, как только функция попытается выполнить `obj[index]`. Интерпретатор Python обнаруживает индексирование последовательностей, выходящее за допустимые пределы, и сообщает о нем *генерацией* встроенного исключения `IndexError`:

```
>>> fetcher(x, 4)              # Стандартный обработчик – интерфейс оболочки
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range

Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
  Файл <stdin>, строка 2, в fetcher
Ошибка индекса: индекс в строке выходит за допустимые пределы
```

Из-за того, что наш код явно не перехватывает такое исключение, оно просачивается на верхний уровень программы и приводит к вызову *стандартного обработчика исключений*, который просто выводит стандартное сообщение об ошибке. К этому месту в книге вы, наверное, уже получили свою долю стандартных сообщений об ошибках. Они содержат сгенерированное исключение и *трассировку стека* — список всех строк и функций, которые были активными на момент возникновения исключения.

Текст сообщения об ошибке здесь был выведен версией Python 3.7; он может слегка варьироваться в зависимости от выпуска и даже от интерактивной оболочки, так что вы не должны полагаться на его точную форму — ни в книге, ни в своем коде. При интерактивном написании кода в базовом интерфейсе оболочки именем файла будет просто `<stdin>`, что обозначает стандартный входной поток.

При работе в интерактивной оболочке с графическим пользовательским интерфейсом IDLE именем файла является `<pyshell>` и отображаются также строки исходного кода. В любом случае номера строк в файле не особо содержательны, когда файла нет (позже в текущей части книги вы увидите более интересные сообщения об ошибках):

```
>>> fetcher(x, 4)              # Стандартный обработчик - интерфейс IDLE
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    fetcher(x, 4)
  File "<pyshell#3>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range

Трассировка (самый последний вызов указан последним):
  Файл <pyshell#5>, строка 1, в <модуль>
    fetcher(x, 4)
  Файл <pyshell#3>, строка 2, в fetcher
    return obj[index]
Ошибка индекса: индекс в строке выходит за допустимые пределы
```

В более реалистичной программе, запущенной вне интерактивной оболочки, после вывода сообщения об ошибке стандартный обработчик верхнего уровня также немедленно *прекращает работу* программы. Такой курс действий имеет смысл для простых сценариев; ошибки часто должны быть фатальными, и лучшее, что вы можете предпринять, когда они возникают — изучить стандартное сообщение об ошибке.

Перехват исключений

Однако временами это не то, что вас интересует. Скажем, серверные программы обычно должны оставаться активными даже после возникновения внутренних ошибок. Если вы не хотите иметь дело со стандартным поведением исключений, тогда поместите вызов внутрь оператора `try`, чтобы самостоятельно перехватывать исключения:

```
>>> try:
...     fetcher(x, 4)
... except IndexError:                 # Перехват и восстановление
...     print('got exception')         # Получено исключение
...
got exception
>>>
```

Теперь интерпретатор Python автоматически переходит на ваш *обработчик* (блок ниже конструкции `except`, в которой указано генерируемое исключение), когда на стадии выполнения блока `try` инициируется исключение. Результатом оказывается вкладывание вложенного блока кода внутрь обработчика ошибок, который перехватывает исключения данного блока.

При интерактивном взаимодействии вроде показанного далее после выполнения конструкции `except` мы возвращаемся обратно в подсказку Python. В более реалистичной программе операторы `try` не только перехватывают исключения, но также осуществляют *восстановление* после них:

```
>>> def catcher():
        try:
            fetcher(x, 4)
        except IndexError:
            print('got exception')     # Получено исключение
        print('continuing')            # Продолжение
>>> catcher()
got exception
continuing
>>>
```

На этот раз после перехвата и обработки исключения программа возобновляет выполнение ниже полного оператора `try`, который его перехватил — вот почему отображается сообщение `continuing`. Мы не видим стандартное сообщение об ошибке, а программа продолжает нормально двигаться своим путем.

Обратите внимание, что в Python отсутствует способ *возвратиться обратно* к коду, который сгенерировал исключение (конечно, не считая повторного запуска кода, достигнувшего данной точки). Как только вы перехватили исключение, поток управления продолжается после полного оператора `try`, перехватившего исключение, но не после оператора, его инициировавшего. На самом деле Python очищает память от любых функций, которые завершили работу в результате возникновения исключения, подобных функции `fetcher` в нашем примере; они не возобновляемы. Оператор `try` перехватывает исключения и является тем местом, где программа возобновляет выполнение.

Замечание по представлению. В этой части для ряда операторов `try` верхнего уровня снова указываются приглашения `...` интерактивной подсказки, т.к. их код не будет работать в случае вырезания и вставки, если только он не вложен в функцию или класс (`except` и другие строки должны быть выровнены с `try` и не иметь добавочных предваряющих пробелов, необходимых для иллюстрации структуры отступов). Для нормального выполнения просто набирайте или вставляйте операторы с приглашениями `...` по одной строке за раз.

Генерация исключений

До сих пор мы позволяли интерпретатору Python генерировать исключения, совершая ошибки (преднамеренно!), но наши сценарии тоже могут инициировать исключения, т.е. исключения могут генерироваться Python или вашей программой и перехватываться или нет. Чтобы инициировать исключение вручную, просто запустите оператор `raise`. Генерируемые пользователем исключения перехватываются тем же способом, что и исключения, которые генерирует интерпретатор Python. Следующий код нельзя считать самым полезным кодом, когда-либо написанным на Python, но он важен тем, что инициирует встроенное исключение `IndexError`:

```
>>> try:
...     raise IndexError                  # Генерация исключения вручную
... except IndexError:
...     print('got exception')            # Получено исключение
...
got exception
```

Как обычно, если генерируемые пользователем исключения не перехватываются, то они распространяются вплоть до стандартного обработчика исключений и прекращают работу программы с выводом стандартного сообщения об ошибке:

```
>>> raise IndexError
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
IndexError

Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
Ошибка индекса
```

Как вы увидите в следующей главе, оператор `assert` тоже может применяться для генерации исключений — он представляет собой условный оператор `raise`, используемый главным образом при отладке на стадии разработки:

```
>>> assert False, 'Nobody expects the Spanish Inquisition!'
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AssertionError: Nobody expects the Spanish Inquisition!

Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
Ошибка утверждения: Никто не ждёт испанскую инквизицию!
```

Исключения, определяемые пользователем

Представленный в предыдущем разделе оператор `raise` генерировал *встроенное* исключение, определенное во встроенной области видимости Python. Как вы узнаете позже в этой части книги, можно также самостоятельно определять новые исключе-

ния, специфичные для ваших программ. Определяемые пользователем исключения реализуются с помощью *классов*, унаследованных от встроенного класса исключения — обычно класса по имени Exception:

```
>>> class AlreadyGotOne(Exception): pass     # Исключение, определяемое
                                             #  пользователем

>>> def grail():
        raise AlreadyGotOne()                # Генерирует экземпляр

>>> try:
...     grail()
... except AlreadyGotOne:                    # Перехват по имени класса
...     print('got exception')               # Получено исключение
...
got exception
>>>
```

В следующей главе будет показано, что конструкция as оператора except может предоставлять доступ к самому объекту исключения. Исключения на основе классов позволяют сценариям формировать категории исключений, которые способны наследовать поведение, а также иметь присоединенную информацию о состоянии и методы. Вдобавок они могут настраивать текст своих сообщений об ошибках, отображаемый в ситуации, когда не был совершен перехват:

```
>>> class Career(Exception):
        def __str__(self): return 'So I became a waiter...'

>>> raise Career()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
__main__.Career: So I became a waiter...
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
__main__.Career: Так я стал официантом...
>>>
```

Действия при завершении

Наконец, операторы try могут содержать слово finally, т.е. иметь в своем составе блоки finally. Они выглядят похожими на обработчики except для исключений, но комбинация try/finally указывает действия при завершении, которые всегда выполняются "на выходе" независимо от того, происходили исключение в блоке try или нет:

```
>>> try:
...     fetcher(x, 3)
... finally:                          # Действия при завершении
...     print('after fetch')          # После извлечения
...
'm'
after fetch
>>>
```

Если блок try завершается без исключения, то блок finally выполнится и программа возобновит работу после полного оператора try. В данном случае наличие оператора try кажется слегка нелепым — мы могли бы просто набрать print сразу после вызова функции и вообще избавиться от try:

```
fetcher(x, 3)
print('after fetch')
```

Тем не менее, здесь присутствует проблема: если вызов функции сгенерирует исключение, тогда поток управления никогда не доберется до `print`. Комбинация `try/finally` позволяет избежать этой ловушки — когда в блоке `try` все же возникает исключение, блоки `finally` выполняются во время раскручивания стека программы:

```
>>> def after():
        try:
            fetcher(x, 4)
        finally:
            print('after fetch')      # После извлечения
        print('after try?')           # После try?

>>> after()
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
  Файл <stdin>, строка 3, в after
  Файл <stdin>, строка 2, в fetcher
Ошибка индекса: индекс в строке выходит за допустимые пределы
>>>
```

Здесь мы не получаем сообщение `after try?`, потому что поток управления не возобновляется после блока `try/finally`, когда возникает исключение. Взамен интерпретатор Python переходит обратно к выполнению действия `finally` и затем *распространяет* исключение вверх к предыдущему обработчику (в этой ситуации к стандартному разработчику на верхнем уровне). Если мы изменим вызов функции `fetcher` так, чтобы не инициировать исключение, то код `finally` по-прежнему выполнится, но программа продолжит выполнения после `try`:

```
>>> def after():
        try:
            fetcher(x, 3)
        finally:
            print('after fetch')
        print('after try?')

>>> after()
after fetch
after try?
>>>
```

На практике комбинации `try/except` удобны для перехвата и восстановления после исключений, а комбинации `try/finally` оказываются полезными, когда требуется гарантия того, что действия при завершении будут запускаться независимо от любых исключений, которые могут возникать в коде блока `try`. Например, вы можете применять `try/except` для перехвата ошибок, инициируемых кодом, который импортируется из сторонней библиотеки, и `try/finally` для обеспечения того, что обращения к функциям закрытия файлов или подключений к серверу всегда выполняются. Несколько практических примеров такого рода приводятся позже в текущей части книги.

Хотя конструкции except и finally служат концептуально отличающимся целям, начиная с Python 2.5, мы можем смешивать их в одном операторе try — конструкция finally выполняется при выходе независимо от того, генерировалось ли исключение, и было ли оно перехвачено конструкцией except.

Как выяснится в следующей главе, линейки Python 2.X и Python 3.X предлагают альтернативу try/finally в случае использования некоторых видов объектов. Оператор with/as запускает логику *управления контекстом* объекта, чтобы гарантировать выполнение действий при завершении безотносительно к любым исключениям в его вложенном блоке:

```
>>> with open('lumberjack.txt', 'w') as file:     # Всегда при выходе
                                                  #   закрывать файл
        file.write('The larch!\n')
```

Несмотря на то что такой вариант требует меньше строк кода, он применим только при обработке определенных объектных типов, поэтому try/finally является более универсальной структурой для завершения и часто проще, чем реализации класса в случаях, где with еще не поддерживается. С другой стороны, with/as может выполнять также действия начального запуска и поддерживает определяемый пользователем код управления контекстами с доступом к полному комплекту инструментов ООП на Python.

Что потребует внимания: проверка на предмет ошибок

Один из способов посмотреть, насколько полезны исключения, предусматривает сравнение кодовых стилей в Python и языках без исключений. Скажем, если вы хотите написать надежную программу на языке С, то обычно должны проверять возвращаемые значения или коды состояния после каждой операции, способной сбиться с пути, и распространять результаты проверок во время выполнения программы:

```
doStuff()
{                                 # Программа на С
    if (doFirstThing() == ERROR)  # Выявлять ошибки повсеместно,
        return ERROR;             # даже если они здесь не обрабатываются
    if (doNextThing() == ERROR)
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}
```

На самом деле реалистичные программы на С часто содержат столько же кода, предназначенного для обнаружения ошибок, сколько и кода для выполнения фактической работы. Но в Python вам не придется быть до такой степени методичными (вплоть до паранойи!). Вы можете взамен помещать произвольно крупные фрагменты программы внутрь обработчиков исключений и просто писать код, делающий действительную работу, предполагая о том, что обычно все будет хорошо:

```
def doStuff():          # Код на Python
    doFirstThing()      # Мы не обязаны здесь заботиться об исключениях,
    doNextThing()       # поэтому нет необходимости и выявлять их
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()       # Тут нас интересуют результаты, так что
    except:             # это единственное место, где требуется проверка
        badEnding()
    else:
        goodEnding()
```

Так как при возникновении исключения управление немедленно передается обработчику, нет нужды снабжать весь код защитой от ошибок, к тому же отсутствуют добавочные накладные расходы в плане производительности, связанные с выполнением всех проверок. Кроме того, поскольку интерпретатор Python выявляет ошибки автоматически, в первую очередь вашему коду часто нет необходимости вообще осуществлять проверки на предмет ошибок. В итоге исключения позволяют почти совершенно игнорировать необычные случаи и избегать написания кода проверки на предмет ошибок, который способен отвлечь от подлинных целей программы.

Резюме

Итак, большая часть истории об исключениях была изложена; исключения – действительно простой инструмент.

Подводя итоги, можно сказать, что исключения Python являются высокоуровневым механизмом управления потоком выполнения. Они могут генерироваться интерпретатором Python либо вашими программами. В обоих случаях исключения допускается игнорировать (для выдачи стандартного сообщения об ошибке) или перехватывать посредством операторов `try` (с целью обработки в вашем коде). Оператор `try` имеет два логических формата, которые начиная с версии Python 2.5, можно объединять – один обрабатывает исключения, а другой выполняет код финализации независимо от того, возникало исключение или нет. Операторы `raise` и `assert` инициируют исключение по требованию – как встроенные, так и новые исключения, определяемые с помощью классов. Оператор `with/as` представляет собой альтернативный способ гарантирования того, что действия при завершении будут выполнены для объектов, которые их поддерживают.

В остатке этой части книги мы рассмотрим ряд деталей о задействованных операторах, исследуем другие виды конструкций, которые могут появляться под `try`, и обсудим объекты исключений, основанные на классах. В следующей главе мы начнем с того, что пристальнее взглянем на введенные здесь операторы. Однако прежде чем двигаться дальше, ответьте на несколько контрольных вопросов.

Проверьте свои знания: контрольные вопросы

1. Назовите три случая, для обработки которых хорошо подходят исключения.
2. Что произойдет с исключением, если вы не предпримете ничего специального для его обработки?
3. Как сценарий может восстанавливаться после исключения?
4. Назовите два способа генерации исключений в сценарии.
5. Назовите два способа указания действий, подлежащих выполнению на стадии завершения вне зависимости от того, возникало исключение или нет.

Проверьте свои знания: ответы

1. Обработка исключений полезна для обработки ошибок, выполнения действий при завершении и уведомления о событиях. Вдобавок она упрощает поддержку особых случаев и может использоваться для реализации альтернативных потоков управления как что-то вроде структурированной операции "безусловного перехода". В целом обработка исключений также сокращает объем кода проверки на предмет ошибок, который может требоваться в программе — из-за того, что все ошибки попадают в обработчики, исчезает необходимость в проверке исхода каждой операции.
2. Любое неперехваченное исключение, в конце концов, просачивается в стандартный обработчик исключений, который Python предоставляет на верхнем уровне программы. Этот обработчик выводит легко узнаваемое сообщение об ошибке и прекращает работу программы.
3. Если вы не хотите, чтобы выводилось стандартное сообщение, а работа программы прекращалась, тогда можете предусмотреть операторы `try/except` для перехвата и восстановления после исключений, которые генерируются внутри их вложенных блоков кода. После того, как исключение перехвачено, оно заканчивается, и программа продолжает выполнение после `try`.
4. Операторы `raise` и `assert` можно применять для генерации исключения в точности, как если бы оно инициировалось самим интерпретатором Python. В принципе исключение можно также сгенерировать, допустив программную ошибку, но обычно это не является прямой целью!
5. Оператор `try/finally` может использоваться для обеспечения того, что действия будут выполнены после выхода из блока кода, невзирая на то, было сгенерировано исключение в блоке или нет. Оператор `with/as` может также применяться для того, чтобы гарантировать выполнение действий при завершении, но только при обработке объектных типов, которые это поддерживают.

ГЛАВА 34

Детали обработки исключений

В предыдущей главе мы кратко взглянули на связанные с исключениями операторы в действии. Здесь мы собираемся копнуть глубже – в этой главе предоставляется более формальное введение в синтаксис Python для обработки исключений. В частности, мы исследуем детали, лежащие в основе операторов `try`, `raise`, `assert` и `with`. Как вы увидите, несмотря на то, что указанные операторы в основном прямолинейны, они предлагают мощные инструменты для работы с исключительными условиями в коде на Python.

Одно заблаговременное процедурное примечание. В последние годы история об исключениях значительным образом менялась. Начиная с Python 2.5, конструкция `finally` может появляться в операторе `try` вместе с конструкциями `except` и `else` (ранее комбинировать их было невозможно). Кроме того, в версиях Python 3.0 и Python 2.6 новый оператор диспетчера контекста `with` стал официальным, и теперь определяемые пользователем исключения должны быть реализованы в виде экземпляров классов, унаследованных от встроенного суперкласса исключения. Вдобавок Python 3.X имеет слегка модифицированный синтаксис для оператора `raise` и конструкций `except`, часть которого доступна в Python 2.6 и 2.7.

Основное внимание в настоящем издании я буду уделять состоянию исключений в последних выпусках Python 2.X и 3.X. Но так как вполне вероятно, что в течение какого-то времени вы по-прежнему будете встречать в коде первоначальные методики, попутно я буду указывать, каким образом все развивалось в данной области.

Оператор `try/except/else`

После ознакомления с основами самое время переходить к деталям. В последующем обсуждении я сначала представлю `try/except/else` и `try/finally` как отдельные операторы, поскольку в версиях, предшествующих Python 2.5, они исполняют разные роли и не могут комбинироваться, а в наши дни отличаются, во всяком случае, логически. Согласно предыдущей врезке "На заметку!" в Python 2.5 и более поздних версиях `except` и `finally` могут смешиваться в одном операторе `try`; последствия такого объединения мы увидим после исследования двух первоначальных форм обособленно.

Синтаксически `try` представляет собой составной оператор, содержащий несколько частей. Он начинается со строки заголовка `try`, за которой следует блок операторов

(обычно) с отступами, затем один или больше конструкций except, идентифицирующих перехватываемые исключения вместе с блоками для их обработки, и в конце необязательная конструкция else с блоком кода. Слова try, except и else связываются друг с другом за счет их отступа на одинаковый уровень (т.е. выравнивания по вертикали). Для справки ниже показан общий и наиболее полный формат в Python 3.X:

```
try:
    операторы                  # Главное действие, выполняемое первым
except имя1:
    операторы                  # Выполняются, если в течение блока try
                               #   сгенерировалось исключение имя1
except (имя2, имя3):
    операторы                  # Выполняются, если произошло любое
                               #   из указанных исключений
except имя4 as переменная:
    операторы                  # Выполняются, если сгенерировалось исключение имя4,
                               #   экземпляр исключения присваивается переменной
except:
    операторы                  # Выполняются, если были сгенерированы
                               #   все остальные исключения
else:
    операторы                  # Выполняются, если в течение блока try исключения
                               #   не генерировались
```

Семантически блок под заголовком try в этом операторе представляет *главное действие* оператора – код, который вы пытаетесь выполнить и помещаете его внутрь логики обработки ошибок. Конструкции except определяют *обработчики* для исключений, генерируемых в течение блока try, а конструкция else (если есть) предоставляет обработчик, подлежащий выполнению, если *никакие* исключения не возникали. Элемент *переменная* относится к характерным особенностям операторов raise и классов исключений, которые мы обсудим более подробно позже в главе.

Как работают операторы try

С точки зрения работы ниже описано, как выполняются операторы try. При входе в оператор try интерпретатор Python запоминает текущий контекст программы, чтобы он мог возвратиться к нему, если возникнет исключение. Первыми выполняются операторы, вложенные внутрь заголовка try. То, что происходит следующим, зависит от того, генерировались ли исключения во время выполнения операторов блока try, и соответствуют ли они тем, которые отслеживает try.

- Если исключение происходит во время выполнения операторов блока try и оно соответствует одному из перечисленных в операторе, тогда интерпретатор Python переходит обратно на try и запускает операторы под первой конструкцией except, дающей совпадение со сгенерированным исключением. Затем объект сгенерированного исключения присваивается переменной, указанной после ключевого слова as в конструкции (при его наличии). После выполнения блока except поток управления возобновляется ниже полного оператора try (если только сам блок except не сгенерирует еще одно исключение, в случае чего процесс начинается заново с этой точки в коде).
- Если исключение происходит во время выполнения операторов блока try, но оно не соответствует одному из перечисленных в операторе, тогда исключение распространяется до следующего самого последнего введенного оператора try,

который дает совпадение с исключением. Если найти такой оператор `try` не удается и поиск достигает верхнего уровня процесса, то интерпретатор Python уничтожает программу и выводит стандартное сообщение об ошибке.

- Если во время выполнения операторов блока `try` никаких исключений не произошло, тогда интерпретатор Python запускает операторы под строкой `else` (при ее наличии) и поток управления возобновляется с места, которое находится ниже полного оператора `try`.

Другими словами, конструкции `except` перехватывают любые совпадающие исключения, которые происходят во время выполнения блока `try`, а конструкция `else` запускается, только когда при выполнении блока `try` исключения не возникают. Сгенерированные исключения *сопоставляются* с исключениями, перечисленными в конструкциях `except`, по отношениям с суперклассами, которые мы исследуем в следующей главе, и пустая конструкция `except` (без имен исключений) соответствует всем (или всем остальным) исключениям.

Конструкции `except` являются *специализированными* обработчиками исключений — они перехватывают исключения, которые возникают только в рамках операторов ассоциированного блока `try`. Однако так как операторы блока `try` иногда вызывают функции, реализованные где-то в другом месте программы, источник исключения может оказаться вне самого оператора `try`.

Фактически блок `try` способен обращаться к произвольно крупным объемам программного кода, в том числе кода, содержащего операторы `try` — при возникновении исключений поиск в таких операторах будет осуществляться первым. То есть операторы `try` могут *вкладываться во время выполнения*; в главе 36 мы рассмотрим эту тему подробнее.

Конструкции оператора `try`

Когда вы пишете оператор `try`, после заголовка `try` могут появляться разнообразные конструкции. В табл. 34.1 приведена сводка по всем возможным формам — вы обязаны использовать хотя бы одну. Некоторые из них мы уже встречали: как известно, конструкции `except` перехватывают исключения, конструкции `finally` запускаются при выходе, а конструкции `else` выполняются, если исключения не возникали.

Таблица 34.1. Формы конструкций оператора `try`

Форма конструкции	Интерпретация
`except:`	Перехватывает все (или все остальные) типы исключений
`except` *`имя`*`:`	Перехватывает только специфическое исключение
`except` *`имя`* `as` *`значение`*`:`	Перехватывает указанное исключение и присваивает его экземпляр
`except (`*`имя1, имя2`*`):`	Перехватывает любое из перечисленных исключений
`except (`*`имя1,имя2`*`) as` *`значение`*`:`	Перехватывает любое из перечисленных исключений и присваивает его экземпляр
`else:`	Выполняется, если в блоке `try` исключения не генерировались
`finally:`	Всегда выполняется при выходе

Формально может присутствовать любое количество конструкций except, но else записывается, только если есть, по крайней мере, одна конструкция except, и допускается только одна else и одна finally. До версии Python 2.4 конструкция finally должна быть одна (без else или except); в действительности try/finally – другой оператор. Тем не менее, начиная с Python 2.5, конструкция finally может появляться в операторе, где есть except и else (правила упорядочения будут более подробно обсуждаться позже в главе, когда мы займемся унифицированным оператором try).

Мы будем исследовать элементы с дополнительной частью as *значение* при рассмотрении оператора raise далее в главе. Они предоставляют доступ к объектам, которые были сгенерированы как исключения.

Перехват любого и всех исключений

Новыми для нас являются первая и четвертая строки в табл. 34.1.

- Конструкции except, не содержащие имен исключений (except:), перехватывают все исключения, которые до того не перечислялись в операторе try.
- Конструкции except, в которых указан список исключений в круглых скобках (except (e1, e2, e3):), перехватывают любое из перечисленных исключений.

Поскольку интерпретатор Python ищет совпадение внутри заданного try путем инспектирования конструкций except *сверху вниз*, версия в круглых скобках имеет такой же эффект, как указание каждого исключения в собственной конструкции except, но блок операторов, ассоциированный с каждым исключением, придется писать только раз. Ниже приведен пример множества конструкций except в работе, который демонстрирует, насколько специфичными могут оказаться ваши обработчики:

```
try:
    action()
except NameError:
    ...
except IndexError:
    ...
except KeyError:
    ...
except (AttributeError, TypeError, SyntaxError):
    ...
else:
    ...
```

Если во время выполнения вызова функции action генерируется исключение, тогда интерпретатор Python возвращается к try и ищет первую конструкцию except, в которой указано имя возникшего исключения. Он инспектирует конструкции except сверху вниз плюс слева направо и запускает операторы под первой конструкцией, давшей совпадение. Если ни одна конструкция не обеспечила совпадение, тогда исключение распространяется после этого оператора try. Обратите внимание, что конструкция else выполняется, только когда в action *никаких* исключений не произошло – она не запускается в случае генерации исключения, для которого отсутствует соответствующая конструкция except.

Перехват всех исключений: пустая конструкция except и Exception

Если вам действительно нужна "всеобъемлющая" конструкция, то подойдет пустая конструкция except:

```
try:
   action()
except NameError:
   ...              # Обработка NameError
except IndexError:
   ...              # Обработка IndexError
except:
   ...              # Обработка всех остальных исключений
else:
   ...              # Обработка случая отсутствия исключений
```

Пустая конструкция except представляет собой своеобразное *групповое* средство — по причине перехвата всего она позволяет вашим обработчикам быть настолько универсальными или специфическими, насколько вы хотите. В некоторых сценариях такая форма может оказаться более удобной, чем перечисление всех возможных исключений в try. Скажем, следующий оператор try перехватывает все, ничего не перечисляя:

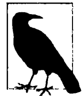
```
try:
   action()
except:
   ...              # Перехват всех возможных исключений
```

Однако пустые конструкции except также привносят ряд вопросов при проектировании. Несмотря на удобство, они могут перехватывать непредвиденные системные исключения, не относящиеся к вашему коду, и неумышленно перехватывать исключения, которые предназначены для другого обработчика. Например, даже системные вызовы для выхода и нажатия комбинаций клавиш <Ctrl+C> генерируют в Python исключения, которые обычно желательно пропускать. Хуже того, пустые конструкции except способны также отлавливать подлинные программные ошибки, для которых вероятно имеет смысл видеть соответствующие сообщения. Мы вернемся к этому вопросу при рассмотрении затруднений в конце данной части книги. Сейчас я просто рекомендую применять их с осторожностью.

В Python 3.X более строго поддерживается альтернатива, которая решает одну из таких проблем — перехват исключения по имени Exception дает почти тот же самый результат, что и пустая конструкция except, но игнорирует исключения, связанные с системными вызовами для выхода:

```
try:
   action()
except Exception:
   ...              # Перехват всех возможных исключений кроме вызовов для
выхода
```

Мы исследуем, как эта форма делает свою магию, в следующей главе, когда приступим к изучению классов исключений. Выражаясь кратко, она работает из-за того, что исключения дают совпадение, если являются подклассами класса, указанного в конструкции except, а Exception представляет собой суперкласс для всех исключений, которые вы должны перехватывать подобным образом. Данная форма обеспечивает почти такое же удобство пустой конструкции except без риска перехвата событий выхода. Несмотря на то что она лучше, с ней связаны те же самые опасности — особенно в плане маскирования программных ошибок.



Примечание, касающееся нестыковки версий. Дополнительные сведения о порции as конструкций except в try также ищите далее при описании оператора raise. Синтаксически Python 3.X требует формы конструкции обработчика except E as V:, перечисленной в табл. 34.1 и используемой в книге, а не более старой формы except E, V:. Последняя форма все еще доступна (но не рекомендуется) в Python 2.6 и 2.7: если она применяется, то преобразуется в первую форму.

Изменение было внесено с целью устранения путаницы касательно двойной роли запятых в более старой форме. В этой форме два альтернативных исключения прекрасно записываются как except (E1, E2):. Поскольку Python 3.X поддерживает только форму as, запятые в конструкции обработчика всегда означают кортеж независимо от того, используются круглые скобки или нет, а значения интерпретируются как альтернативные исключения, подлежащие перехвату.

Тем не менее, как будет показано далее, такой вариант не изменяет правила областей видимости в Python 2.X: даже с новым синтаксисом as в Python 2.X переменная V по-прежнему доступна после блока except. В Python 3.X переменная V позже не будет доступной и на самом деле принудительно удаляется.

Конструкция else оператора try

Целевое назначение конструкции else не всегда сразу очевидно для новичков в Python. Однако без нее отсутствует прямой способ сообщить (без установки и проверки булевских флагов), продолжил поток управления выполнение после оператора try из-за того, что никаких исключений не возникало или же исключение произошло и обработано. В любом случае мы оказываемся после оператора try:

```
try:
    ...выполнить код...
except IndexError:
    ...обработать исключение...
# Мы сюда попали из-за того, что try потерпел неудачу или же прошел?
```

Во многом подобно тому, как конструкции else в циклах придают причине выхода бóльшую очевидность, конструкция else предоставляет в операторе try синтаксис, который делает то, что произошло, ясным и недвусмысленным:

```
try:
    ...выполнить код...
except IndexError:
    ...обработать исключение...
else:
    ...исключения не возникали...
```

Вы можете *почти полностью* смоделировать конструкцию else, переместив ее код в блок try:

```
try:
    ...выполнить код...
    ...исключения не возникали...
except IndexError:
    ...обработать исключение...
```

Тем не менее, такой прием может привести к некорректной классификации исключения. Если действие "исключения не возникали" сгенерирует экземпляр IndexError,

то он будет зарегистрирован как отказ блока try и ошибочно запустит обработчик исключений ниже try (тонко, но верно!). За счет применения взамен явной конструкции else вы делаете логику более очевидной и гарантируете, что обработчики except будут запускаться только для реальных отказов в коде, помещенном внутрь try, а не для отказов в действии для случая "исключения не возникали" конструкции else.

Пример: стандартное поведение

Поскольку поток управления программы легче объяснять с помощью Python, чем на естественном языке, давайте рассмотрим несколько примеров, которые дополнительно проиллюстрируют основы исключений в контексте более крупных фрагментов кода из файлов.

Я уже упоминал, что исключения, не перехваченные операторами try, проникают на верхний уровень процесса Python и выполняют стандартную логику обработки исключений Python (т.е. интерпретатор Python прекращает работу функционирующей программы и выводит стандартное сообщение об ошибке). В целях демонстрации запуск следующего файла модуля bad.py приводит к генерированию исключения, связанного с делением на ноль:

```
def gobad(x, y):
    return x / y

def gosouth(x):
    print(gobad(x, 0))

gosouth(1)
```

Так как программа игнорирует инициируемое ею исключение, интерпретатор Python уничтожает ее и выводит сообщение:

```
% python bad.py
Traceback (most recent call last):
  File "C:\Code\bad.py", line 7, in <module>
    gosouth(1)
  File "C:\Code\bad.py", line 5, in gosouth
    print(gobad(x, 0))
  File "C:\Code\bad.py", line 2, in gobad
    return x / y
ZeroDivisionError: division by zero
Трассировка (самый последний вызов указан последним):
  Файл "C:\Code\bad.py", строка 7, в <модуль>
    gosouth(1)
  Файл "C:\Code\bad.py", строка 5, в gosouth
    print(gobad(x, 0))
  Файл "C:\Code\bad.py", строка 2, в gobad
    return x / y
Ошибка деления на ноль: деление на ноль
```

Файл модуля bad.py выполнялся в окне командной оболочки Python 3.X. Сообщение состоит из трассировки стека (Traceback) и имени сгенерированного исключения вместе с сопутствующими деталями. В трассировке стека перечислены все строки, которые были активными, когда возникло исключение, от старых к новым. Обратите внимание, что поскольку мы работаем не в интерактивном сеансе, в данном случае информация об имени файла и номере строки более полезна. Скажем, здесь мы видим, что деление на ноль произошло в последней записи трассировки — в строке 2 файла bad.py, т.е. операторе return [1].

[1] Как упоминалось в предыдущей главе, текст сообщений об ошибках и трассировок стека имеет тенденцию варьироваться с течением времени и в зависимости от оболочки. Не беспокойтесь, если ваши сообщения об ошибках не полностью совпадают с приведенными в книге. Например, когда пример запускается в оболочке IDLE из Python 3.7, текст сообщения об ошибках содержит имена файлов с полными абсолютными путями.

Поскольку Python обнаруживает и сообщает обо всех ошибках во время выполнения за счет генерирования исключений, исключения тесно связаны с идеями обработки ошибок и отладки в целом. Если вы прорабатывали рассматриваемые в книге примеры, тогда в ходе дела, несомненно, сталкивались с одним или двумя исключениями — даже опечатки обычно генерируют SyntaxError или другое исключение, когда файл импортируется или выполняется (т.е. при запуске компилятора). По умолчанию вы получаете полезное отображение ошибки, подобное только что показанному, которое помогает отследить проблему.

Часто такое стандартное сообщение об ошибке — это все, что вам нужно для устранения проблем в коде. Для решения более сложных задач отладки вы можете перехватывать исключения посредством операторов try либо использовать один из инструментов отладки, которые были представлены в главе 3 первого тома и будут подытожены в главе 36, например, стандартный библиотечный модуль pdb.

Пример: перехват встроенных исключений

Стандартная обработка исключений нередко оказывается именно тем, что вас интересует — особенно в случае кода из файла сценария верхнего уровня, где ошибка зачастую должна немедленно прекращать работу программы. Для многих программ предпринимать в коде что-то более специфичное касательно ошибок нет никакой необходимости.

Однако иногда вам захочется перехватывать ошибки и восстанавливаться после них. Если нежелательно, чтобы ваша программа прекращала работу, когда интерпретатор Python генерирует исключение, тогда просто перехватите ее, поместив программную логику внутрь оператора try. Это важная возможность для таких программ, как сетевые серверы, которые обязаны функционировать постоянно. Скажем, в следующем коде из файла kaboom.py производится перехват и восстановление после ошибки TypeError, генерируемой интерпретатором Python сразу же после того, как вы попытаетесь выполнить конкатенацию списка и строки (вспомните, что операция + ожидает с обеих сторон последовательности того же самого типа):

```
def kaboom(x, y):
    print(x + y)                # Генерируется TypeError

try:
    kaboom([0, 1, 2], 'spam')
except TypeError:               # Перехват и восстановление
    print('Hello world!')
print('resuming here')          # Продолжение здесь независимо от того,
                                #    было исключение или нет
```

Когда в функции kaboom возникает исключение, поток управления переходит на конструкцию except оператора try, которая выводит сообщение. Поскольку после такого перехвата исключение становится "недействующим", программа продолжает выполнение ниже try, а не прекращается интерпретатором Python. В сущности, код обрабатывает и очищает ошибку, восстанавливая ваш сценарий:

```
% python kaboom.py
Hello world!
resuming here
```

Имейте в виду, что как только ошибка перехвачена, поток управления возобновляет выполнение с места, где вы ее перехватили (т.е. после `try`); не существует прямого способа перейти обратно к тому месту, где исключение возникло (здесь в функции `kaboom`). В известном смысле исключения больше похожи на простые переходы, чем на вызовы функций — возвратиться в код, который сгенерировал ошибку, возможности нет.

Оператор try/finally

Другая разновидность оператора `try` является специализацией, имеющей отношение к действиям финализации (она же завершение). Если в состав `try` входит конструкция `finally`, тогда интерпретатор Python будет всегда запускать ее блок операторов "при выходе" из оператора `try` независимо от того, возникало исключение во время выполнения блока `try` или нет. Вот общая форма:

```
try:
    операторы               # Это действие выполняется первым
finally:
    операторы               # Этот код всегда выполняется при выходе
```

В этом варианте интерпретатор Python начинает с запуска блока операторов, ассоциированных с заголовком `try`, как обычно. То, что происходит следующим, зависит от того, возникало ли исключение во время выполнения блока `try`.

- Если во время выполнения блока `try` исключение не возникало, тогда интерпретатор Python переходит к выполнению блока `finally` и продолжает выполнение с места после оператора `try`.
- Если во время выполнения блока `try` возникло исключение, то Python по-прежнему выполняет блок `finally`, но затем исключение распространяется до ранее пройденного оператора `try` или до стандартного обработчика исключений; программа не возобновляет выполнение с места ниже конструкции `finally` оператора `try`. То есть блок `finally` запускается, даже если генерируется исключение, но в отличие от `except` конструкция `finally` не останавливает исключение — после выполнения блока `finally` оно продолжает быть сгенерированным.

Форма `try/finally` удобна, когда вы хотите иметь полную уверенность в том, что действие произойдет после выполнения определенного кода независимо от поведения программы, касающегося исключений. На практике она позволяет указывать действия по очистке, которые должны происходить всегда, такие как закрытие файлов и разрыв соединений с сервером, когда они требуются.

Обратите внимание, что в Python 2.4 и предшествующих версиях конструкция `finally` не может применяться в операторе `try`, где есть `except` и `else`, а потому в случае использования более старого выпуска вариант `try/finally` лучше считать отдельной формой оператора. Тем не менее, в Python 2.5 и последующих версиях `finally` может появляться вместе с `except` и `else`, поэтому в наши дни действительно существует единственный оператор `try` с множеством необязательных конструкций (что мы вскоре обсудим). Однако какую бы версию вы не применяли, конструкция `finally` по-прежнему служит той же самой цели — указание действия по "очистке", которые должны выполняться всегда независимо от любых исключений.

Как будет показано далее в главе, начиная с версий Python 2.6 и Python 3.0, новый оператор `with` и его диспетчеры контекстов предлагают основанный на объектах способ выполнения похожей работы для действий при выходе. В отличие от `finally` этот новый оператор также поддерживает действия при входе, но его сфера ограничена объектами, которые реализуют используемый им протокол диспетчеров контекстов.

Пример: написание кода действий при завершении с помощью try/finally

В предыдущей главе приводилось несколько простых примеров `try/finally`. Ниже приведен более реалистичный пример, который иллюстрирует типичную роль данного оператора:

```
class MyError(Exception): pass

def stuff(file):
    raise MyError()

file = open('data', 'w')  # Открытие выходного файла
                          #   (также может потерпеть неудачу)
try:
    stuff(file)           # Генерирует исключение
finally:
    file.close()          # Всегда закрывать файл, чтобы сбросить буферы
вывода
print('not reached')      # Выполнение продолжается здесь, только если
                          #   не было исключений
```

Когда функция `stuff` генерирует свое исключение, поток управления переходит к выполнению блока `finally`, чтобы закрыть файл. Затем исключение распространяется либо до еще одного `try`, либо до стандартного обработчика верхнего уровня, который выводит стандартное сообщение об ошибке и прекращает работу программы. Следовательно, оператор после этого `try` никогда не будет достигнут. Если функция `stuff` *не* сгенерирует исключение, то программа все же выполнит блок `finally` для закрытия файла, но затем продолжит работу ниже полного оператора `try`.

В этом особом случае мы поместили вызов функции обработки файла внутрь оператора `try` с конструкцией `finally`, чтобы обеспечить закрытие файла и потому финализацию независимо от того, генерирует функция исключение или нет. Таким образом, последующий код может быть уверен в том, что содержимое буфера вывода файла сбрасывается из памяти на диск. Аналогичная кодовая структура может гарантировать, что подключения к серверу закрываются и т.п.

Как известно из главы 9 первого тома, файловые объекты автоматически закрываются при сборке мусора в стандартном Python (CPython), что очень полезно для временных файловых объектов, которые не присваивались переменным. Тем не менее, не всегда легко спрогнозировать, когда произойдет сборка мусора, особенно в крупных программах или альтернативных реализациях Python с отличающимися политиками сборки мусора (скажем, Jython, PyPy). Оператор `try` делает закрытие файлов более явным и принадлежащим специфическому блоку кода. Он гарантирует, что файл будет закрыт при выходе из блока безотносительно к тому, произошло исключение или нет.

Функция из рассмотренного примера не так уж и полезна (она всего лишь генерирует исключение), но помещение вызовов внутрь операторов `try/finally` является хорошим способом обеспечения того, что ваши действия по закрытию при заверше-

нии всегда выполняются. И снова интерпретатор Python всегда запускает код в блоках finally независимо от того, возникало исключение в блоке try или нет [1].

Обратите внимание на то, что определяемое пользователем исключение здесь снова реализовано с помощью *класса* – как будет более формально описано в главе 35, в Python 2.6, 3.0 и последующих выпусках в обеих линейках все исключения обязаны быть экземплярами классов.

Унифицированный оператор try/except/finally

Во всех версиях, предшествующих Python 2.5 (приблизительно для 15 лет его существования), оператор try поступал в двух разновидностях, которые на самом деле были двумя отдельными операторами. Мы могли применять либо finally, гарантируя, что код очистки всегда выполняется, либо записывать блоки except для перехвата и восстановления после специфических исключений и дополнительно указывать конструкцию else, подлежащую выполнению, если исключения не возникали.

Другими словами, конструкция finally не могла смешиваться с except и else. Отчасти причиной были проблемы реализации, а частично то, что смысл их смешивания казался неясным – перехват и восстановление после исключений выглядело несовместимой концепцией с выполнением действий очистки.

Тем не менее, в Python 2.5 и последующих версиях два оператора объединены. В наши дни мы можем смешивать конструкции finally, except и else в том же самом операторе – отчасти из-за похожей полезности в языке Java. То есть теперь оператор можно записывать в такой форме:

```
try:                          # Объединенная форма
    главное-действие
except Exception1:
    обработчик1
except Exception2:            # Перехват исключений
    обработчик2
...
else:                         # Обработчик для случая отсутствия исключений
    блок-else
finally:                           # finally охватывает все остальное
    блок-finally
```

Первым выполняется код в блоке *главное-действие* оператора, как обычно. Если этот код генерирует исключение, тогда друг за другом проверяются все блоки except в поисках совпадения со сгенерированным исключением. Если было сгенерировано исключение Exception1, то выполняется блок *обработчик1*; если Exception2, то блок *обработчик2* и т.д. Если никакие исключения не генерировались, тогда выполняется *блок-else*.

Независимо от того, что произошло ранее, *блок-finally* выполняется один раз по завершении блока *главное-действие* и после обработки любых сгенерированных исключений. Фактически код в *блок-finally* будет выполняться, даже когда в обработчике исключений или в *блок-else* возникла ошибка и сгенерировано новое исключение.

---

[1] Конечно, если только не случится аварийный отказ интерпретатора Python. Однако он неплохо работает, чтобы избежать этого, проверяя на предмет всех возможных ошибок в ходе выполнения программы. Когда программа действительно терпит неудачу, то обычно из-за дефекта в связанном коде расширений C за рамками Python.

Как всегда, конструкция `finally` не заканчивает исключение — если исключение активно во время выполнения *блок-finally*, оно продолжает распространяться после того, как *блок-finally* выполнен, а поток управления переходит куда-то в другое место внутри программы (в еще один `try` или в стандартный обработчик верхнего уровня). Если при выполнении блока `finally` активных исключений нет, то поток управления возобновляет работу после полного оператора `try`.

Совокупный эффект в том, что блок `finally` выполняется всегда, не считаясь со следующими условиями:

- в главном действии произошло исключение, которое было обработано;
- в главном действии произошло исключение, которое не было обработано;
- в главном действии исключения не возникали;
- в одном из обработчиков было сгенерировано новое исключение.

Опять-таки конструкция `finally` предназначена для указания действий очистки, которые всегда должны совершаться при выходе из `try`, независимо от того, какие исключения были сгенерированы или обработаны.

Унифицированный синтаксис оператора `try`

При сочетании подобного рода оператор `try` обязан иметь либо `except`, либо `finally`, а порядок его частей должен выглядеть примерно так:

```
try -> except -> else -> finally
```

где конструкции `else` и `finally` необязательны, может быть ноль и более конструкций `except`, но при наличии `else` должна присутствовать хотя бы одна конструкция `except`. Вообще говоря, оператор `try` состоит из двух частей: конструкции `except` с необязательной конструкцией `else` и/или конструкции `finally`.

На самом деле синтаксическую форму объединенного оператора можно описать более точно следующим образом (квадратные скобки обозначают необязательные части, а символ звездочки — наличие нуля и более конструкций):

```
try:                                # Формат 1
    операторы
except [тип [as значение]]:         # [тип [, значение]] в Python 2.X
    операторы
[except [тип [as значение]]:
    операторы]*
[else:
    операторы]
[finally:
    операторы]
try:                                # Формат 2
    операторы
finally:
    операторы
```

В соответствии с этими правилами конструкция `else` может появляться только при наличии, по меньшей мере, одной конструкции `except`, вдобавок всегда допускается смешивать `except` и `finally` независимо от того, имеется `else` или нет. Можно также смешивать `finally` и `else`, но только при наличии `except` (хотя `except` разрешено опускать имя исключения, чтобы перехватывать все и запускать описанный позже оператор `raise` для повторной генерации текущего исключения). Если вы на-

рушите любое из приведенных правил упорядочения, тогда до запуска вашего кода Python сгенерирует исключение, связанное с синтаксической ошибкой.

Комбинирование `finally` и `except` за счет вложения

До выхода Python 2.5 фактически было возможно комбинировать конструкции `finally` и `except` в `try`, синтаксически вкладывая `try/except` внутрь блока `try` оператора `try/finally`. Мы исследуем эту методику более полно в главе 36, но ее основы помогут прояснить смысл комбинированного оператора `try` — следующий код дает тот же самый результат, что и новая объединенная форма, показанная в начале раздела:

```
try:                              # Вложенный эквивалент объединенной формы
    try:
        главное-действие
    except Exception1:
        обработчик1
    except Exception2:
        обработчик2
    ...
    else:
        ошибки-отсутствуют
finally:
    очистка
```

И снова блок `finally` всегда запускается при выходе независимо от того, что происходило в главном действии и какие обработчики исключений выполнялись во вложенном операторе `try` (отследите приведенные ранее четыре сценария и убедитесь, что здесь все работает одинаково). Поскольку `else` всегда требует `except`, такая вложенная форма даже поддерживает те же самые ограничения смешивания, присущие унифицированной форме оператора, которые были кратко описаны в предыдущем разделе.

Однако вложенный эквивалент некоторым представляется менее ясным и требует большего объема кода, чем новая объединенная форма — несмотря на всего лишь одну добавочную строку из четырех символов плюс отступы. Смешивание `finally` в том же самом операторе значительно облегчает написание и восприятие кода и в наши дни является предпочитаемой методикой.

Пример унифицированного оператора `try`

Ниже предлагается демонстрация работы объединенной формы оператора `try`. В файле `mergedexc.py` реализованы четыре распространенных сценария с операторами `print`, которые описывают содержание каждого:

```
# Файл mergedexc.py (Python 3.X + 2.X)
sep = '-' * 45 + '\n'

# Исключение генерируется и перехватывается
print(sep + 'EXCEPTION RAISED AND CAUGHT')
try:
    x = 'spam'[99]
except IndexError:
    print('except run')           # выполняется except
finally:
    print('finally run')          # выполняется finally
print('after run')                # после выполнения
```

```
# Исключения не генерируются
print(sep + 'NO EXCEPTION RAISED')
try:
   x = 'spam'[3]
except IndexError:
   print('except run')        # выполняется except
finally:
   print('finally run')       # выполняется finally
print('after run')            # после выполнения

# Исключения не генерируются, с конструкцией else
print(sep + 'NO EXCEPTION RAISED, WITH ELSE')
try:
   x = 'spam'[3]
except IndexError:
   print('except run')        # выполняется except
else:
   print('else run')          # выполняется else
finally:
   print('finally run')       # выполняется finally
print('after run')            # после выполнения

# Исключение генерируется, но не перехватывается
print(sep + 'EXCEPTION RAISED BUT NOT CAUGHT')
try:
   x = 1 / 0
except IndexError:
   print('except run')        # выполняется except
finally:
   print('finally run')       # выполняется finally
print('after run')            # после выполнения
```

Запуск этого кода в версии Python 3.7 дает показанный далее вывод; в Python 2.X поведение кода и вывод будут такими же, потому что каждый вызов `print` выводит одиночный элемент, но сообщение об ошибке слегка отличается. Отследите код, чтобы понять, каким образом обработка исключений производит вывод каждого из четырех тестов:

```
c:\code> py -3 mergedexc.py
-----------------------------------------------
EXCEPTION RAISED AND CAUGHT
except run
finally run
after run
-----------------------------------------------
NO EXCEPTION RAISED
finally run
after run
-----------------------------------------------
NO EXCEPTION RAISED, WITH ELSE
else run
finally run
after run
-----------------------------------------------
```

```
EXCEPTION RAISED BUT NOT CAUGHT
finally run
Traceback (most recent call last):
  File "C:\Code\mergedexc.py", line 39, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero
Трассировка (самый последний вызов указан последним):
  Файл "C:\Code\mergedexc.py", строка 39, в <модуль>
    x = 1 / 0
Ошибка деления на ноль: деление на ноль
```

В главном действии примера используются встроенные операции для генерирования (или нет) исключений, и он полагается на тот факт, что в ходе выполнения кода интерпретатор Python всегда делает проверки на предмет ошибок. В следующем разделе объясняется, как генерировать исключения вручную.

Оператор raise

Чтобы генерировать исключения явно, можно записывать операторы `raise`. Их общая форма проста – оператор `raise` состоит из слова `raise`, за которым дополнительно указывается класс или экземпляр класса генерируемого исключения:

```
raise экземпляр   # Генерирует экземпляр класса
raise класс       # Создает и генерирует экземпляр класса: создает экземпляр
raise             # Повторно генерирует самое последнее исключение
```

Как упоминалось ранее, в Python 2.6, 3.0 и последующих версиях исключения всегда являются экземплярами классов. Таким образом, первая форма `raise` считается наиболее распространенной – мы напрямую предоставляем *экземпляр*, созданный либо перед `raise`, либо внутри самого оператора `raise`. Если взамен мы передаем *класс*, тогда интерпретатор Python вызывает конструктор класса без аргументов для создания экземпляра исключения, подлежащего генерации; такая форма эквивалентна добавлению круглых скобок к ссылке на класс. Последняя форма повторно генерирует самое последнее исключение; она обычно применяется в обработчиках исключений для распространения исключений, которые были перехвачены.

Примечание, касающееся нестыковки версий. В Python 3.X больше не поддерживается форма `raise` *`Exc`*, *`Args`*, которая по-прежнему доступна в Python 2.X. Вместо нее используйте в Python 3.X форму с вызовом для создания экземпляра `raise` *`Exc`*`(`*`Args`*`)`, описанную в настоящей книге. Эквивалентная форма с запятой в Python 2.X является унаследованным синтаксисом, который предоставляется для совместимости с теперь исчезнувшей моделью исключений на основе строк и объявлен устаревшим в Python 2.X. В случае применения форма с запятой преобразуется в форму с вызовом Python 3.X.

Как и в более ранних выпусках, форма `raise` *`Exc`* также позволяет указывать имя класса – в обеих линейках она преобразуется в `raise` *`Exc`*`()`, вызывая конструктор класса без аргументов. Помимо своего исчезнувшего синтаксиса с запятой оператор `raise` в Python 2.X также допускает исключения на основе строк или классов, но первый из двух вариантов удален в Python 2.6, объявлен устаревшим в Python 2.5 и подробно здесь не рассматривается, а лишь кратко упоминается в следующей главе. В наши дни для новых исключений используйте классы.

Генерация исключений

Чтобы сделать все более понятным, давайте обратимся к нескольким примерам. Со встроенными исключениями показанные далее две формы эквивалентны – обе генерируют экземпляр указанного класса исключения, но первая создает экземпляр неявно:

```
raise IndexError               # Класс (экземпляр создается неявно)
raise IndexError()             # Экземпляр (создается явно в операторе)
```

Мы также можем создать экземпляр заблаговременно – поскольку оператор `raise` принимает любой вид ссылки на объект, следующие два примера генерируют `IndexError` в точности как предыдущие два:

```
exc = IndexError()             # Создать экземпляр заблаговременно
raise exc

excs = [IndexError, TypeError]
raise excs[0]
```

Когда генерируется исключение, интерпретатор Python отправляет вместе с ним сгенерированный экземпляр. Если оператор `try` содержит конструкцию `except` *имя* `as` *X*:, тогда переменной *X* будет присвоен экземпляр, доставленный `raise`:

```
try:
    ...
except IndexError as X:  # X присваивается объект сгенерированного экземпляра
    ...
```

Ключевое слово `as` в обработчике `try` необязательно (если оно опущено, то экземпляр просто не присваивается имени), но его добавление дает обработчику возможность доступа к данным экземпляра и методам в классе исключения.

Эта модель работает и для определяемых пользователем исключений, реализованных посредством классов – скажем, вот пример передачи аргумента конструктору класса исключения, который становится доступным в обработчике через присвоенный экземпляр:

```
class MyExc(Exception): pass
...
raise MyExc('spam')            # Класс исключения с аргументом конструктора
...
try:
   ...
except MyExc as X:             # Атрибуты экземпляра, доступные в обработчике
    print(X.args)
```

Тем не менее, мы не будем посягать здесь на тему следующей главы, где и будут представлены дальнейшие детали.

Независимо от того, как вы указываете исключения, они всегда идентифицируются *объектами экземпляров* классов, и любой заданный момент времени может быть активным самое большее один такой объект. Будучи однажды перехваченным конструкцией `except` где-нибудь в программе, исключение исчезает (т.е. не будет передаваться еще одному оператору `try`), если только оно не генерируется повторно другим оператором `raise` или возникшей ошибкой.

Области видимости и переменные except в try

Мы исследуем объекты исключений более глубоко в следующей главе. Однако теперь, увидев переменную as в действии, мы можем окончательно прояснить специфичную к версиям проблему с областями видимости, которая была подытожена в главе 17 первого тома. В *Python 2.X* имя переменной со ссылкой на исключение в конструкции except *не* локализуется внутри самой конструкции и доступно после выполнения ассоциированного блока:

```
c:\code> py -2
>>> try:
...     1 / 0
... except Exception as X:    # Python 2.X не локализует X в любом случае
...     print X
...
integer division or modulo by zero
целочисленное деление или деление по модулю на ноль
>>> X
ZeroDivisionError('integer division or modulo by zero',)
```

Это верно в Python 2.X независимо от того, применяется стиль as из Python 3.X или более ранний синтаксис с запятой:

```
>>> try:
...     1 / 0
... except Exception, X:
...     print X
...
integer division or modulo by zero
целочисленное деление или деление по модулю на ноль
>>> X
ZeroDivisionError('integer division or modulo by zero',)
```

И напротив, *Python 3.X* локализует имя переменной со ссылкой на исключение внутри блока except — после выхода из блока переменная не будет доступной почти как временная переменная цикла в выражениях включений Python 3.X (как отмечалось ранее, Python 3.X также не воспринимает в except синтаксис с запятыми из Python 2.X):

```
c:\code> py -3
>>> try:
...     1 / 0
... except Exception, X:
SyntaxError: invalid syntax
Синтаксическая ошибка: недопустимый синтаксис

>>> try:
...     1 / 0
... except Exception as X:  # Python 3.X локализует имена as внутри блока except
...     print(X)
...
division by zero
деление на ноль
>>> X
NameError: name 'X' is not defined
Ошибка в имени: имя X не определено
```

Тем не менее, в отличие от переменных цикла в выражениях включений в Python 3.X после выхода из блока except эта переменная *удаляется*. Так происходит оттого, что в противном случае она сохранила бы ссылку на стек вызовов времени выполнения, которая отложила бы сборку мусора, оставив выделенным избыточное пространство памяти. Однако удаление переменной происходит, даже если вы используете имя где-то в другом месте, и является более крайней политикой, чем применяемая для включений:

```
>>> X = 99
>>> try:
...     1 / 0
... except Exception as X:    # Python 3.X локализует переменную X
                              #   _и_ удаляет ее при выходе!
...     print(X)
...
division by zero
деление на ноль
>>> X
NameError: name 'X' is not defined
Ошибка в имени: имя X не определено
>>> X = 99
>>> {X for X in 'spam'}     # Python 2.X/3.X только локализует X, но не удаляет
{'s', 'a', 'p', 'm'}
>>> X
99
```

Из-за этого вы обычно должны использовать в конструкциях except оператора try уникальные имена переменных, хотя они и локализуются внутри областей видимости. Если вам необходимо ссылаться на экземпляр исключения после оператора try, тогда просто присвойте его еще одному имени, которое не будет автоматически удаляться:

```
>>> try:
...     1 / 0
... except Exception as X:    # Python удаляет эту ссылку
...     print(X)
... Saveit = X                # Присвоить экземпляр исключения для его
                              #   сохранения при необходимости
...
division by zero
деление на ноль
>>> X
NameError: name 'X' is not defined
Ошибка в имени: имя X не определено
>>> Saveit
ZeroDivisionError('division by zero',)
```

Распространение исключений с помощью raise

Оператор raise несколько более функционален, чем мы видели до сих пор. Например, оператор raise, который не содержит имени исключения или добавочного значения данных, просто повторно генерирует текущее исключение. Такая форма обычно применяется, когда исключение необходимо перехватить и обработать, но его исчезновение в коде нежелательно:

```
>>> try:
...     raise IndexError('spam')    # Исключения запоминают аргументы
... except IndexError:
...     print('propagating')        # распространение
...     raise             # Повторная генерация самого последнего исключения
...
propagating
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: spam
распространение
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 2, в <модуль>
Ошибка индекса: spam
```

Выполнение `raise` подобным способом повторно генерирует исключение и передает его более высокому обработчику (или стандартному обработчику верхнего уровня, который выводит стандартное сообщение об ошибке и останавливает программу). Обратите внимание на отображение в сообщении об ошибке аргумента, переданного классу исключения; вы узнаете, почему так происходит, в следующей главе.

Сцепление исключений в Python 3.X: `raise from`

Исключения иногда могут генерироваться в ответ на другие исключения — как преднамеренно, так и по причине новых ошибок в программе. Для поддержки полноценного обнаружения в таких случаях в Python 3.X (но не в Python 2.X) операторам `raise` также разрешено иметь дополнительную конструкцию `from`:

```
raise новое-исключение from другое-исключение
```

Когда конструкция `from` используется в явном запросе `raise`, следующее за `from` выражение указывает еще один класс или экземпляр для присоединения к атрибуту `__cause__` нового генерируемого исключения. Если сгенерированное исключение не перехвачено, тогда интерпретатор Python выводит оба исключения как часть стандартного сообщения об ошибке:

```
>>> try:
...     1 / 0
... except Exception as E:
...     raise TypeError('Bad') from E    # Явно сцепленные исключения
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: Bad
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 2, в <модуль>
Ошибка деления на ноль: деление на ноль

Вышеприведенное исключение было непосредственной причиной следующего исключения:

Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 4, в <модуль>
Ошибка типа: Bad
```

Когда из-за ошибки в программе внутри обработчика исключений неявно генерируется исключение, автоматически соблюдается похожая процедура: предыдущее исключение присоединяется к атрибуту `__context__` нового исключения и снова отображается в стандартном сообщении об ошибке, если оно не было перехвачено:

```
>>> try:
...     1 / 0
... except:
...     badname    # Неявно сцепленные исключения
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'badname' is not defined
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 2, в <модуль>
Ошибка деления на ноль: деление на ноль

Во время обработки вышеприведенного исключения возникло еще одно исключение:

Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 2, в <модуль>
Ошибка в имени: имя badname не определено
```

В обоих случаях из-за того, что объекты первоначальных исключений, присоединенные к объектам новых исключений, *сами* могут иметь присоединенные причины, цепочка причинно-следственной зависимости способна быть *произвольно длинной* и полностью отображаться в сообщениях об ошибках. То есть сообщения об ошибках могут содержать сведения о более чем двух исключениях. Совокупный эффект в явном и неявном контексте состоит в том, что программисты знают все вовлеченные исключения, когда одно исключение инициирует другое:

```
>>> try:
...     try:
...         raise IndexError()
...     except Exception as E:
...         raise TypeError() from E
... except Exception as E:
...     raise SyntaxError() from E
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
TypeError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
SyntaxError: None
Трассировка (самый последний вызов указан последним):
```

```
  Файл "<stdin>", строка 3, в <модуль>
Ошибка индекса

Вышеприведенное исключение было непосредственной причиной следующего исключения:

Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 5, в <модуль>
Ошибка типа

Вышеприведенное исключение было непосредственной причиной следующего исключения:

Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 7, в <модуль>
Синтаксическая ошибка: None
```

Код вроде показанного ниже схожим образом отображает три исключения, несмотря на неявную генерацию:

```
try:
    try:
        1 / 0
    except:
        badname
except:
    open('nonesuch')
```

Как и унифицированный оператор `try`, сцепленные исключения полезны подобно их аналогам в других языках (в том числе Java и C#), хотя неясно, из каких языков они были позаимствованы. В Python исключения все еще являются не вполне понятным расширением, поэтому за дополнительными сведениями обращайтесь в руководства по языку. На самом деле согласно следующей врезке "На заметку!" в версии Python 3.3 появился способ *подавления* вывода сцепленных исключений.

Подавление вывода сцепленных исключений в Python 3: `raise from None`. В Python 3.3 была представлена новая форма синтаксиса — указание `None` для имени исключения в операторе `raise from`:

```
raise новое-исключение from None
```

Она позволяет запретить отображение контекста сцепленных исключений, описанного в предыдущем разделе. В итоге сообщения об ошибках становятся менее загроможденными в приложениях, которые выполняют преобразования между типами исключений наряду с обработкой цепочек исключений.

Оператор `assert`

В качестве довольно особого случая для целей отладки в Python предусмотрен оператор `assert`. По большей части `assert` представляет собой синтаксическое сокращение для распространенной схемы применения `raise` и может считаться *условным* оператором `raise`. Оператор вида:

```
assert test, data    # Часть data является необязательной
```

работает аналогично следующему коду:

```
if __debug__:
    if not test:
        raise AssertionError(data)
```

Другими словами, если в результате вычисления *test* получается False, тогда интерпретатор Python генерирует исключение: элемент *data* (если предоставлен) используется как аргумент для конструктора класса исключения. Подобно всем исключениям AssertionError прекратит выполнение программы, если не перехватить его с помощью try, и элемент *data* будет отображаться в виде части стандартного сообщения об ошибке.

Кроме того, операторы assert могут быть удалены из байт-кода скомпилированной программы в случае применения флага командной строки -O компилятора Python, что оптимизирует программу. AssertionError является встроенным исключением, а флаг __debug__ — встроенным именем, которое автоматически устанавливается в True, если только не используется флаг -O. Применяйте командную строку вроде python -O main.py для запуска программы в оптимизированном режиме и отключения (а потому пропуска) утверждений.

Пример: улавливание нарушений ограничений (но не ошибок!)

Утверждения обычно используются для того, чтобы контролировать соблюдение условий в программах на стадии разработки. В отображаемые ими сообщения об ошибках автоматически включается информация о строке исходного кода и значение, указанное в операторе assert. Возьмем файл asserter.py:

```
def f(x):
    assert x < 0, 'x must be negative'    # x должно быть отрицательным
    return x ** 2
```

```
% python
>>> import asserter
>>> asserter.f(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    asserter.f(1)
  File "C:\Code\asserter.py", line 2, in f
    assert x < 0, 'x must be negative'
AssertionError: x must be negative
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 1, в <модуль>
    asserter.f(1)
  Файл "C:\Code\asserter.py", строка 2, в f
    assert x < 0, 'x must be negative'
Ошибка утверждения: x должно быть отрицательным
```

Важно помнить о том, что оператор assert предназначен главным образом для улавливания нарушений ограничений, определяемых пользователем, а не для перехвата подлинных программных ошибок. Поскольку интерпретатор Python самостоятельно отлавливает программные ошибки, обычно нет никакой необходимости записывать операторы assert для перехвата таких вещей, как индексы, выходящие за допустимые границы, несовпадения типов и деление на ноль:

```
def reciprocal(x):
    assert x != 0   # В целом бесполезное утверждение!
    return 1 / x    # Python автоматически проверяет на предмет равенства нулю
```

Такие сценарии применения `assert`, как правило, излишни – из-за того, что интерпретатор Python генерирует исключения при возникновении ошибок автоматически, вы с тем же успехом могли бы позволить ему делать эту работу за вас. Как правило, вам не нужно делать явную проверку на предмет ошибок в собственном коде.

Разумеется, у большинства правил существуют отклонения – как предполагалось ранее в книге, если функция должна выполнить длительные или невосстанавливаемые действия, прежде чем она доберется до места, где будет сгенерировано исключение, то вам по-прежнему может требоваться проверка на предмет ошибок. Тем не менее, даже в этом случае будьте осторожны, чтобы не сделать проверки чрезмерно специфическими или ограничивающими, иначе вы снизите полезность своего кода.

Еще один распространенный сценарий использования `assert` демонстрировался в примере абстрактного суперкласса в главе 29; там оператор `assert` применялся для того, чтобы вызовы неопределенных методов терпели неудачу с выводом сообщения. Это редкий, но полезный инструмент.

Диспетчеры контекстов `with/as`

В версиях Python 2.6 и Python 3.0 был введен новый оператор, связанный с исключениями – `with` вместе с его необязательной конструкцией `as`. Оператор `with/as` предназначен для работы с объектами *диспетчеров контекстов*, которые поддерживают новый основанный на методах протокол, по духу похожий на способ работы с методами итерационных инструментов в протоколе итерации. Данная возможность доступна как вариант в версии Python 2.5, но должна быть включена посредством оператора `import` вида:

```
from __future__ import with_statement
```

Оператор `with` также подобен оператору `using` в языке C#. Хотя тема диспетчеров контекстов, вообще говоря, необязательна и ориентирована на сложные инструменты (а потому является кандидатом на рассмотрение в следующей части книги), диспетчеры контекстов достаточно легковесны и полезны, чтобы объединить их здесь с остатком инструментального комплекта для работы с исключениями.

Выражаясь кратко, оператор `with/as` задуман как альтернатива распространенной идиоме использования `try/finally`; подобно `try/finally` оператор `with` в значительной степени предназначен для указания действий стадии завершения или "очистки", которые должны выполняться независимо от того, возникало ли исключение в течение шага обработки.

В отличие от `try/finally` оператор `with` основан на объектном протоколе для указания действий, подлежащих выполнению вокруг блока кода. Это делает оператор `with` менее универсальным, квалифицирует его как избыточный в ролях с завершением и требует реализации классов для объектов, которые не поддерживают его протокол. С другой стороны, `with` также поддерживает действия при входе, способен сократить размер кода и позволяет управлять контекстами кода с помощью полноценного ООП.

Python расширяет диспетчерами контекстов ряд встроенных инструментов, таких как файлы, которые автоматически закрываются, и потоки, которые автоматически блокируются и деблокируются, но программисты могут реализовывать собственные диспетчеры контекстов с применением классов. Давайте кратко рассмотрим оператор и его неявный протокол.

Базовое использование

Ниже показан базовый формат оператора `with` с необязательной частью в квадратных скобках:

```
with выражение [as переменная]:
    блок-with
```

Здесь предполагается, что *выражение* возвращает объект, поддерживающий протокол управления контекстами (который вскоре будет обсуждаться). Этот объект может также возвращать значение, которое будет присвоено имени *переменная* при наличии необязательной конструкции `as`.

Обратите внимание, что *переменной* не обязательно присваивается *результат выражения*; результатом *выражения* является объект, который поддерживает протокол управления контекстами, и *переменной* может быть присвоено что-то другое, предназначенное для применения внутри оператора. Затем объект, возвращенный *выражением*, может выполнить код запуска перед началом *блока-with*, а также код завершения после окончания *блока-with* независимо от того, генерировалось ли в блоке исключение.

Некоторые встроенные объекты Python были дополнены поддержкой протокола управления контекстами, а потому могут использоваться с оператором `with`. Скажем, файловые объекты (описанные в главе 9 первого тома) имеют диспетчер контекста, который автоматически закрывает файл после блока `with` безотносительно к тому, генерировалось ли исключение, и независимо от того, если или когда версия Python, выполняющая код, может закрыть его автоматически:

```
with open(r'C:\misc\data') as myfile:
    for line in myfile:
        print(line)
        ...дополнительный код...
```

Вызов `open` возвращает простой файловый объект, который присваивается имени `myfile`. Мы можем применять `myfile` с обычными файловыми инструментами — в данном случае файловый итератор читает строка за строкой в цикле `for`.

Однако этот объект поддерживает протокол управления контекстами, используемый оператором `with`. После выполнения такого оператора `with` механизм управления контекстами гарантирует, что файловый объект, на который ссылается `myfile`, автоматически закрывается, даже если во время обработки файла в цикле `for` возникло исключение.

Хотя файловые объекты могут автоматически закрываться при сборке мусора, не всегда легко узнать, когда она случится, особенно когда применяются альтернативные реализации Python. Оператор `with` в такой роли представляет собой альтернативу, которая обеспечивает нам уверенность в том, что закрытие произойдет после выполнения специфического блока кода.

Как было показано ранее, мы можем достичь похожего эффекта с помощью более универсального и явного оператора `try/finally`, но в рассматриваемой ситуации он требует три дополнительных строки административного кода (четыре вместо только одной):

```
myfile = open(r'C:\misc\data')
try:
    for line in myfile:
        print(line)
        ...дополнительный код...
finally:
    myfile.close()
```

Мы не раскрываем в книге модули многопоточной обработки Python (ищите сведения о них в книгах прикладного уровня наподобие *Programming Python* (http://www.oreilly.com/catalog/9780596158101)). Тем не менее, определяемые ими объекты блокировок и условной синхронизации также могут использоваться с оператором with, потому что они поддерживают протокол управления контекстами — в этом случае добавление действий входа и выхода вокруг блока:

```
lock = threading.Lock()                  # После import threading
with lock:
    # критический раздел кода
    ...доступ к разделяемым ресурсам...
```

Здесь механизм управления контекстами гарантирует, что блокировка будет автоматически получена перед выполнением блока и освобождена после окончания блока независимо от исходов исключений.

Как было представлено в главе 5 первого тома, модуль decimal также применяет диспетчеры контекстов для упрощения сохранения и восстановления текущего контекста десятичных чисел, который указывает точность и характеристики округления для вычислений:

```
with decimal.localcontext() as ctx:      # После import decimal
    ctx.prec = 2
    x = decimal.Decimal('1.00') / decimal.Decimal('3.00')
```

После выполнения данного оператора состояние диспетчера текущего потока автоматически восстанавливается в то, которое было до начала оператора. Чтобы сделать то же самое посредством try/finally, нам пришлось бы сохранять контекст перед и восстанавливать его вручную после вложенного блока.

Протокол управления контекстами

Хотя некоторые встроенные типы снабжены диспетчерами контекстов, мы также можем создавать новые такие диспетчеры самостоятельно. Для реализации диспетчеров контекстов классы используют специальные методы, которые относятся к категории методов перегрузки операций и позволяют взаимодействовать с оператором with. Интерфейс, ожидаемый от применяемых в операторах with объектов, довольно сложен, и большинству программистов необходимо лишь знать, как использовать существующие диспетчеры контекстов. Однако у разработчиков инструментов может возникнуть потребность в написании новых диспетчеров контекстов, специфичных для приложений, поэтому давайте бегло взглянем, что им предстоит делать.

Вот как в действительности работает оператор with.

1. Выражение вычисляется, давая в результате объект *диспетчера контекста*, который обязан иметь методы `__enter__` и `__exit__`.

2. Вызывается метод `__enter__` диспетчера контекста. Возвращаемое им значение присваивается переменной в конструкции as при ее наличии либо попросту отбрасывается.

3. Выполняется код во вложенном блоке with.

4. Если в блоке with возникает исключение, тогда вызывается метод `__exit__(type, value, traceback)` с передачей ему деталей исключения. Это те же самые значения, которые возвращает функция `sys.exc_info`, описанная в руководствах по Python и позже в данной части книги. Если метод `__exit__`

возвращает значение False, тогда исключение генерируется повторно; иначе оно заканчивается. Обычно исключение должно быть сгенерировано заново, чтобы оно распространилось за пределы оператора with.

5. Если в блоке with исключение не возникало, то метод __exit__ все равно вызывается, но для всех его аргументов *type*, *value* и *traceback* передаются None.

Рассмотрим небольшую иллюстрацию протокола в действии. В файле withas.py с показанным далее содержимым определяется объект диспетчера контекста, который отслеживает вход и выход из блока with в любом операторе with, где он применяется:

```
class TraceBlock:
    def message(self, arg):
        print('running ' + arg)                          # выполнение
    def __enter__(self):
        print('starting with block')                     # начало блока
        return self
    def __exit__(self, exc_type, exc_value, exc_tb):
        if exc_type is None:
          print('exited normally\n')                     # нормальный выход
        else:
          print('raise an exception! ' + str(exc_type))  # генерация исключения
            return False                                 # Распространение
if __name__ == '__main__':
    with TraceBlock() as action:
        action.message('test 1')
        print('reached')                                 # достигнуто
    with TraceBlock() as action:
        action.message('test 2')
        raise TypeError
        print('not reached')                             # не достигнуто
```

Обратите внимание, что метод __exit__ класса TraceBlock возвращает False, чтобы исключение распространялось; удаление оператора return дало бы тот же самый эффект, т.к. стандартное возвращаемое значение None функций по определению равно False. Кроме того, метод __enter__ возвращает self в качестве объекта для присваивания переменной as; в других сценариях использования может возвращаться совершенно другой объект.

Во время выполнения диспетчер контекста отслеживает вход и выход из блока операторов with посредством своих методов __enter__ и __exit__. Ниже приведена демонстрация сценария в работе под управлением Python 3.X или 2.X (как обычно, в Python 2.X вывод слегка отличается, и сценарий допускает запуск в Python 2.6, 2.7 и 2.5 при включенной возможности):

```
c:\code> py -3 withas.py
starting with block
running test 1
reached
exited normally

starting with block
running test 2
raise an exception! <class 'TypeError'>
Traceback (most recent call last):
  File "withas.py", line 22, in <module>
    raise TypeError
TypeError
```

```
Трассировка (самый последний вызов указан последним):
  Файл "C:\Code\ withas.py", строка 22, в <модуль>
    raise TypeError
Ошибка типа
```

Диспетчеры контекстов также могут задействовать информацию о состоянии и наследование ООП, но являются относительно сложным механизмом, ориентированным на разработчиков инструментов, поэтому мы опускаем здесь остальные детали (за исчерпывающим описанием обращайтесь к стандартным руководствам по Python — например, существует библиотечный модуль `contextlib`, который предлагает дополнительные инструменты для реализации диспетчеров контекстов). Для более простых целей оператор `try/finally` обеспечивает достаточную поддержку выполнения действий при завершении без необходимости в написании классов.

Множество диспетчеров контекстов в Python 3.1, 2.7 и последующих версиях

В версии Python 3.1 было введено расширение `with`, которое со временем появилось и в Python 2.7. В этих и последующих версиях Python в операторе `with` можно также указывать множество (иногда называемых "вложенными") диспетчеров контекстов с помощью нового синтаксиса в виде запятой. Скажем, в показанном далее коде действия выхода обоих файловых объектов автоматически запускаются при выходе из блока операторов независимо от исходов исключений:

```
with open('data') as fin, open('res', 'w') as fout:
    for line in fin:
        if 'some key' in line:
            fout.write(line)
```

Допускается указывать любой количество диспетчеров контекстов и множество элементов работают аналогично вложенным операторам `with`. В версиях Python, поддерживающих такую возможность, следующий код:

```
with A() as a, B() as b:
    ...операторы...
```

эквивалентен приведенному ниже коду, который также работает в Python 3.0 и 2.6:

```
with A() as a:
    with B() as b:
        ...операторы...
```

Дополнительные подробности можно найти в пояснительных записках к выпуску Python 3.1, но здесь мы кратко взглянем на данное расширение в действии. Чтобы реализовать параллельный просмотр двух файлов по строкам, в представленном далее коде применяется оператор `with` для открытия двух файлов и объединения их строк посредством `zip` без необходимости в ручном закрытии по завершении (предполагая, что оно обязательно):

```
>>> with open('script1.py') as f1, open('script2.py') as f2:
...     for pair in zip(f1, f2):
...         print(pair)
...
('# A first Python script\n', 'import sys\n')
('import sys                  # Load a library module\n', 'print(sys.path)\n')
('print(sys.platform)\n', 'x = 2\n')
('print(2 ** 32)              # Raise 2 to a power\n', 'print(x ** 32)\n')
```

Такую кодовую структуру можно использовать, например, для построчного *сравнения* двух текстовых файлов, заменив print оператором if и вызвав enumerate для номеров строк:

```
with open('script1.py') as f1, open('script2.py') as f2:
    for (linenum, (line1, line2)) in enumerate(zip(f1, f2)):
        if line1 != line2:
            print('%s\n%r\n%r' % (linenum, line1, line2))
```

Тем не менее, предыдущий прием не так уж полезен в CPython, потому что входные файлы не требуют сбрасывания буферов и при освобождении занимаемой памяти файловые объекты автоматически закрываются, если они остались открытыми. В CPython код для параллельного просмотра можно упростить и занимаемая файловыми объектами память станет освобождаться немедленно:

```
for pair in zip(open('script1.py'), open('script2.py')): # Тот же результат,
                                                         # автоматическое закрытие
    print(pair)
```

С другой стороны, из-за отличий в сборщиках мусора альтернативные реализации наподобие PyPy и Jython могут требовать более прямого закрытия внутри циклов во избежание истощения системных ресурсов. Следующий код автоматически закрывает выходной файл, когда оператор заканчивается, чтобы любой буферизированный текст незамедлительно перемещался на диск:

```
>>> with open('script2.py') as fin, open('upper.py', 'w') as fout:
...     for line in fin:
...         fout.write(line.upper())
...
>>> print(open('upper.py').read())
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

В обоих случаях мы можем взамен просто открывать файлы в отдельных операторах и при необходимости закрывать их после обработки, а в некоторых сценариях, вероятно, мы даже должны поступать так — нет никакого смысла применять операторы, перехватывающие исключение, если это означает, что программа все равно не работает!

```
fin = open('script2.py')
fout = open('upper.py', 'w')
for line in fin:                    # Тот же результат, что и в предыдущем коде,
                                    #   автоматическое закрытие
    fout.write(line.upper())
```

Однако в случае, если программы должны продолжать работу после возникновения исключений, формы with также неявно перехватывают исключения и тем самым позволяют избежать написания оператора try/finally в ситуациях, когда закрытие обязательно. Эквивалентная реализация без with оказывается более явной, но требует заметно большего объема кода:

```
fin = open('script2.py')
fout = open('upper.py', 'w')
try:            # Тот же результат, но явное закрытие при возникновении ошибки
    for line in fin:
        fout.write(line.upper())
```

```
finally:
    fin.close()
    fout.close()
```

С другой стороны, try/finally является одиночным инструментом, применимым ко всем сценариям финализации, тогда как with добавляет второй инструмент, который может быть более компактным, но применяется только к определенным объектным типам и удваивает необходимую базу знаний для программистов. Как обычно, вам придется самостоятельно проанализировать все компромиссы.

Резюме

В главе мы рассматривали обработку исключений более подробно за счет исследования операторов Python, связанных с исключениями: try для перехвата, raise для генерации, assert для условной генерации и with для помещения блоков кода внутрь диспетчеров контекстов, которые задают действия входа и выхода.

До сих пор исключения, возможно, казались довольно легковесным инструментом и на самом деле таковыми они и являются; единственная сложность в том, как они идентифицируются. В следующей главе наше исследование продолжается описанием того, как реализовать сами объекты исключений; вы увидите, что классы позволяют создавать новые исключения, специфичные к разрабатываемым программам. Но сначала закрепите пройденный материал главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Для чего предназначен оператор try?
2. Назовите две распространенных вариации оператора try.
3. Для чего предназначен оператор raise?
4. Для чего предназначен оператор assert, и на какой другой оператор он похож?
5. Для чего предназначен оператор with/as, и на какой другой оператор он похож?

Проверьте свои знания: ответы

1. Оператор try осуществляет перехват и восстановление после исключений — он указывает блок кода, подлежащий запуску, и один или большее число обработчиков для исключений, которые могут возникнуть во время выполнения этого блока.

2. Двумя распространенными вариациями оператора try являются try/except/else (для перехвата исключений) и try/finally (для указания действий очистки, которые должны происходить независимо от того, генерировалось исключение или нет). До Python 2.4 они были отдельными операторами, которые можно было комбинировать путем синтаксического вложения; в Python 2.5 и последующих версиях блоки except и finally могут смешиваться в одном операторе, поэтому обе формы оператора были объединены. В объединенной форме finally по-прежнему выполняется при выходе из try, не обращая внимания на то, какие исключения могли быть сгенерированы или обработаны. Фактически объ-

единенная форма эквивалентна вложению `try/except/else` в `try/finally`, а две вариации все еще исполняют логически отличающиеся роли.

3. Оператор `raise` генерирует (инициирует) исключение. Интерпретатор Python внутренне генерирует встроенные исключения при возникновении ошибок, но ваши сценарии с помощью `raise` тоже могут генерировать встроенные или определяемые пользователем исключения.
4. Оператор `assert` генерирует исключение `AssertionError`, если условие оказывается ложным. Он работает подобно условному оператору `raise`, помещенному внутрь оператора `if`, и может быть отключен с использованием флага `-O`.
5. Оператор `with/as` предназначен для автоматизации действий при запуске и завершении, которые должны происходить вокруг блока кода. Он примерно похож на оператор `try/finally` в том, что его действия выхода выполняются независимо от того, возникало исключение или нет, но делает возможным более развитый протокол на основе объектов для указания действий входа *и* выхода, а также способен сократить размер кода. Тем не менее, оператор `with/as` не настолько универсальный, т.к. он применим только к объектам, которые поддерживают его протокол; оператор `try` охватывает гораздо больше сценариев использования.

ГЛАВА 35

Объекты исключений

До сих пор я преднамеренно не давал определение того, чем фактически *является* исключение. Как подсказывалось в предыдущей главе, начиная с версий Python 2.6 и 3.0, встроенные и определяемые пользователем исключения идентифицируются *объектами экземпляров классов*. Именно они генерируются и распространяются в ходе обработки исключений и представляют собой источник класса, который сопоставляется с исключениями, указанными в операторах.

Хотя это означает, что для определения новых исключений в своих программах вы должны использовать ООП – и вводит зависимость от сведений, полное раскрытие которых было отложено до текущей части книги, – базирование исключений на классах и ООП сулит несколько перечисленных ниже преимуществ.

- **Исключения на основе классов организованы по категориям.** Исключения, реализованные в виде классов, поддерживают будущие изменения за счет предоставления категорий – добавление новых исключений впоследствии обычно не требует внесения изменений в операторы `try`.
- **Исключения на основе классов обладают информацией состояния и поведением.** Классы исключений предлагают естественное местоположение для хранения информации о контексте и инструменты для применения в обработчике `try` – экземпляры имеют доступ к присоединенной информации состояния и вызываемым методам.
- **Исключения на основе классов поддерживают наследование.** Основанные на классах исключения могут принимать участие в иерархиях наследования, чтобы получать и настраивать общее поведение – например, унаследованные методы отображения способны предоставлять общий внешний вид для сообщений об ошибках.

Благодаря описанным преимуществам исключения на основе классов хорошо поддерживают развитие программ и более крупные системы. Как обнаружится, по указанным выше причинам все встроенные исключения идентифицируются классами и организуются в дерево наследования. Вы можете делать то же самое с собственными исключениями, определяемыми пользователем.

Фактически в Python 3.X исследуемые здесь встроенные исключения оказываются неотъемлемой составляющей новых исключений, которые вы определяете. Поскольку Python 3.X требует, чтобы определяемые пользователем исключения наследовались от встроенных суперклассов исключений, предлагающих полезные стандартные методы для вывода и сохранения состояния, задача реализации исключений, определяемых

пользователем, также предусматривает понимание ролей, исполняемых встроенными исключениями.

Примечание, касающееся нестыковки версий. В Python 2.6, 3.0 и последующих версиях исключения должны определяться посредством классов. Вдобавок в Python 3.X классы исключений обязаны быть производными от встроенного класса исключения `BaseException`, либо напрямую, либо косвенно. Как мы увидим, в большинстве программ реализуется наследование от подкласса `Exception` данного класса, чтобы поддерживать универсальные обработчики для нормальных типов исключений — его указание в обработчике обеспечит перехват всего, что должно перехватываться. Автономным классическим классам Python 2.X также разрешено служить в качестве исключений, но, как и в Python 3.X, классы нового стиля обязаны быть унаследованными от встроенных классов исключений.

Исключения: назад в будущее

Когда-то давно (до версий Python 2.6 и 3.0) исключения можно было определять двумя разными способами, что усложняло операторы `try`, операторы `raise` и язык Python в целом. В наши дни существует только один способ их определения. И это хорошо: из языка исчез значительный объем хлама, накопленного ради обратной совместимости. Однако поскольку старый способ помогает объяснить, почему исключения стали такими, какими они есть сегодня, и потому, что невозможно полностью забыть историю чего-либо, используемого примерно миллионом людей в течение почти двух десятилетий, давайте начнем наше исследование настоящего с краткого экскурса в прошлое.

Строковые исключения канули в лету!

До выхода версий Python 2.6 и 3.0 исключения можно было определять с помощью экземпляров классов и строковых объектов. В версии Python 2.5 для исключений на основе строк начали выдаваться предупреждения об устаревании, а в версиях Python 2.6 и 3.0 они были удалены, поэтому в настоящее время вы должны применять исключения на основе классов, как показано в книге. Тем не менее, в случае работы с унаследованным кодом вы все еще можете сталкиваться со строковыми исключениями. Они также могут встречаться в книгах, руководствах и веб-ресурсах, созданных несколько лет тому назад (что в годах Python считается вечностью!).

Строковые исключения было легко использовать — подходила любая строка и они сопоставлялись по объектной идентичности, а не по значению (т.е. применялась операция `is`, не `==`):

```
C:\code> C:\Python25\python
>>> myexc = "My exception string"    # Мы были настолько молодыми?...
>>> try:
...     raise myexc
... except myexc:
...     print('caught')
...
caught
```

Такую форму исключений удалили, потому что для более крупных программ и сопровождения кода она была не настолько хорошей, как классы. В современных версиях Python строковые исключения сами генерируют исключения:

```
C:\code> py -3
>>> raise 'spam'
TypeError: exceptions must derive from BaseException
Ошибка типа: исключения должны быть унаследованными от BaseException

C:\code> py -2
>>> raise 'spam'
TypeError: exceptions must be old-style classes or derived from
BaseException, ...etc
Ошибка типа: исключения должны быть классами старого стиля или
унаследованными от BaseException, ...и т.д.
```

Несмотря на невозможность использования строковых исключений в наши дни, в действительности они обеспечивают естественную связку для представления модели исключений на основе классов.

Исключения на основе классов

Строки служили простым способом определения исключений. Однако, как было описано ранее, классы имеют ряд дополнительных преимуществ, которые заслуживают краткого обзора. Самое примечательное то, что они позволяют идентифицировать *категории* исключений, которые оказываются более гибкими в применении и сопровождении, чем простые строки. Кроме того, классы естественным образом позволяют присоединять информацию об исключениях и поддерживают наследование. Поскольку классы рассматриваются многими как лучший подход, теперь они являются обязательными.

Если оставить в стороне детали кода, то главное отличие между строковыми исключениями и исключениями на основе классов связано со способом генерации исключений и их сопоставления с конструкциями `except` в операторах `try`.

- Строковые исключения сравнивались по простой объектной идентичности: сгенерированное исключение сопоставлялось с конструкциями `except` посредством проверки `is`.
- Исключения на основе классов сопоставляются по отношениям с суперклассами: сгенерированное исключение соответствует конструкции `except`, если в ней присутствует класс экземпляра исключения или любой из его суперклассов.

То есть, когда в конструкции `except` оператора `try` указан суперкласс, она перехватывает экземпляры этого суперкласса, а также всех его подклассов ниже в дереве классов. Совокупный эффект состоит в том, что основанные на классах исключения естественным образом поддерживают построение *иерархий* исключений: суперклассы становятся названиями категорий, а подклассы — специфичными видами исключений внутри категории. За счет указания имени общего суперкласса конструкция `except` способна перехватывать целую категорию исключений — любой специфический подкласс будет давать соответствие.

В строковых исключениях концепция подобного рода отсутствовала: из-за того, что они сопоставлялись по простой объектной идентичности, не существовало прямого способа организации исключений в более гибкие категории или группы. В итоге обработчики исключений были связаны с наборами исключений таким образом, что затрудняли внесение изменений.

Кроме идеи с категориями исключения на основе классов эффективнее поддерживают *информацию состояния* исключений (присоединяемую к экземплярам) и позволяют исключениям участвовать в *иерархиях наследования* (чтобы получать общие линии поведения). Поскольку они обеспечивают все преимущества классов и ООП в целом, то предлагают более мощную альтернативу ныне несуществующей модели исключений на основе строк в обмен на скромный объем добавочного кода.

Реализация классов исключений

Давайте рассмотрим пример, чтобы выяснить, как исключения на основе классов воплощаются в коде. В файле `classexc.py`, содержимое которого приведено ниже, мы определяем суперкласс по имени `General` и два подкласса с именами `Specific1` и `Specific2`. В примере иллюстрируется понятие категорий исключений – `General` представляет собой название категории, а два его подкласса являются специфичными типами исключений внутри категории. Обработчики, которые перехватывают суперкласс `General`, будут перехватывать также любые его подклассы, в том числе `Specific1` и `Specific2`:

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0():
    X = General()          # Генерирует экземпляр суперкласса
    raise X

def raiser1():
    X = Specific1()        # Генерирует экземпляр подкласса
    raise X

def raiser2():
    X = Specific2()        # Генерирует экземпляр другого подкласса
    raise X

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General: # Соответствует суперклассу General или любому его подклассу
        import sys
        print('caught: %s' % sys.exc_info()[0])

C:\code> python classexc.py
caught: <class '__main__.General'>
caught: <class '__main__.Specific1'>
caught: <class '__main__.Specific2'>
```

Код в основном прямолинеен, но есть несколько моментов, которые следует отметить.

Суперкласс `Exception`

Классы, используемые для построения деревьев категорий исключений, предъявляют очень мало требований – фактически в примере они по большей части пусты, с телами, в которых ничего не делается, а присутствует только `pass`. Тем не менее, обратите внимание на наследование класса верхнего уровня от встроенного класса `Exception`. В Python 3.X поступать так обязательно; в Python 2.X автономным классическим классам также разрешено служить в качестве исключений, но классы нового стиля должны быть производными от встроенных классов исключений в точности как в Python 3.X. Хотя мы здесь это не задейс-

твуем, потому что суперкласс Exception предоставляет ряд полезных линий поведения, с которыми мы встретимся позже, неплохая идея наследовать от него в любой версии Python.

Генерация экземпляров

В коде мы обращаемся к классам с целью создания *экземпляров* для операторов raise. В рамках модели исключений, основанных на классах, мы всегда генерируем и перехватываем объект экземпляра класса. Если мы указываем в raise имя класса без круглых скобок, тогда интерпретатор Python вызывает конструктор класса, лишенный аргументов, чтобы создать для нас экземпляр. Экземпляры исключений могут создаваться перед raise, как делалось здесь, или внутри самого оператора raise.

Перехват категорий

Код содержит функции, которые генерируют экземпляры всех трех классов в виде исключений, а также оператор try верхнего уровня, вызывающий функции и перехватывающий исключения General. Тот же самый оператор try перехватывает и два специфичных исключения из-за того, что они являются подклассами General — членами его категории.

Детали исключений

В данном случае обработчик исключений применяет вызов sys.exc_info — как будет более подробно обсуждаться в следующей главе, именно так мы можем захватывать самое недавнее исключение в обобщенной манере. Выражаясь кратко, первый элемент в его результате представляет собой класс сгенерированного исключения, а второй — действительный экземпляр. В универсальной конструкции except вроде показанной здесь, которая перехватывает все классы в категории, sys.exc_info является одним способом выяснить, что в точности произошло. В рассмотренной конкретной ситуации вызов sys.exc_info эквивалентен извлечению атрибута __class__ экземпляра. В следующей главе мы увидим, что схема с sys.exc_info также часто используется с конструкциями except, которые перехватывают все исключения.

Последний пункт заслуживает дальнейшего объяснения. Когда исключение перехватывается, мы можем быть уверены в том, что сгенерированный экземпляр принадлежит классу, указанному в except, или одному из его более специфичных подклассов. По этой причине атрибут __class__ экземпляра также дает тип исключения. Скажем, показанный далее вариант из файла classexc2.py работает так же, как предыдущий пример — в нем применяется расширение as конструкции except для присваивания переменной фактически сгенерированного экземпляра:

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0(): raise General()
def raiser1(): raise Specific1()
def raiser2(): raise Specific2()

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General as X:                    # X - сгенерированный экземпляр
        print('caught: %s' % X.__class__)   # То же, что и sys.exc_info()[0]
```

Поскольку `__class__` можно использовать подобным образом для выяснения конкретного типа сгенерированного исключения, вызов `sys.exc_info` более удобен в пустых конструкциях `except`, которые по-другому не могут получить доступ к экземпляру или его классу. Кроме того, более реалистичные программы обычно вообще *не должны заботиться* о том, какое конкретно исключение было сгенерировано — вызывая методы экземпляра класса исключения обобщенным образом, мы автоматически инициируем линию поведения, которая была настроена для сгенерированного исключения.

Более подробно об этом и `sys.exc_info` пойдет речь в следующей главе; в главе 29 и в целом в части VI приведены сведения об атрибуте `__class__` в экземпляре, а в предыдущей главе — обзор расширения `as`.

Для чего используются иерархии исключений?

Так как в примере из предыдущего раздела было только три возможных исключения, он не позволяет надлежащим образом оценить полезность исключений на основе классов. Фактически мы могли бы достичь того же результата, записав в конструкции `except` список имен исключений в круглых скобках:

```
try:
    func()
except (General, Specific1, Specific2):    # Перехватывать любое из них
    ...
```

Такой подход работает и для исчезнувшей модели строковых исключений. Однако для крупных или глубоких иерархий исключений может быть легче перехватывать категории, указывая классы, чем перечислять все члены категории в одиночной конструкции `except`. Пожалуй, более важно то, что по мере развития нужд программ вы можете расширять иерархии исключений, добавляя новые подклассы и не нарушая работу существующего кода.

Предположим для примера, что вы реализуете библиотеку численных расчетов на Python, ориентированную на применение многими программистами. Во время написания кода библиотеки вы идентифицируете две ситуации, когда что-то может пойти не так с числами — деление на ноль и числовое переполнение. Вы документируете их как два автономных исключения, которые ваша библиотека может генерировать:

```
# mathlib.py

class Divzero(Exception): pass
class Oflow(Exception): pass

def func():
    ...
    raise Divzero()
...и так далее...
```

Теперь при использовании вашей библиотеки программисты будут помещать вызовы функций или обращения к классам в операторы `try`, перехватывающие два упомянутых исключения; в конце концов, если они не перехватят ваши исключения, тогда исключения из библиотеки прекратят выполнение их кода:

```
# client.py

import mathlib

try:
    mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow):
    ...выполнить обработку и восстановление...
```

Все работает нормально, и многие программисты начинают эксплуатировать вашу библиотеку. Тем не менее, спустя полгода вы пересматриваете ее (как склонны поступать все программисты!). Попутно вы идентифицируете новую ситуацию, когда что-то может пойти не так — скажем, потерю значимости — и добавляете ее как новое исключение:

```
# mathlib.py

class Divzero(Exception): pass
class Oflow(Exception): pass
class Uflow(Exception): pass
```

К сожалению, после повторного выпуска кода вы создадите проблему с сопровождением для своих пользователей. Если они явно перечисляли ваши исключения, тогда им придется возвратиться к своему коду и внести изменения во всех местах, где производилось обращение к библиотеке, чтобы указать имя вновь добавленного исключения:

```
# client.py

try:
    mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow, mathlib.Uflow):
    ...выполнить обработку и восстановление...
```

Конечно, это не стоит считать концом света. Если ваша библиотека применяется только внутри организации, то вы можете внести изменения самостоятельно. Вы могли бы также поставлять сценарий на Python, который попытается автоматически скорректировать код подобного рода (вероятно, он будет состоять из нескольких десятков строк и окажется правильным решением, во всяком случае, иногда). Однако если многим программистам придется модифицировать все их операторы `try` каждый раз, когда вы изменяете свой набор исключений, то это будет не особенно изящная политика обновления.

Ваши пользователи могут попробовать избежать данной ловушки, записывая пустые конструкции `except` для перехвата *всех* возможных исключений:

```
# client.py

try:
    mathlib.func(...)
except:       # Перехватывать здесь все исключения (или суперкласс Exception)
    ...выполнить обработку и восстановление...
```

Но этот обходной путь может перехватывать больше, чем предполагалось. В ситуациях вроде нехватки памяти, прерываний с клавиатуры (<Ctrl+C>), выходов в систему и даже опечаток в коде их собственного блока `try` будут возникать исключения, которые должны пропускаться, а не перехватываться и ошибочно трактоваться как ошибки из библиотеки. Указание суперкласса `Exception` улучшает положение, но он по-прежнему перехватывает и потому может маскировать ошибки в программе.

И действительно, в таком сценарии пользователи хотят перехватывать и восстанавливаться *только* после конкретных исключений, которые определены и документированы в плане генерирования. Если во время обращения к библиотеке возникает любое другое исключение, то можно ожидать, что оно отражает подлинный дефект в библиотеке (и вероятно повод связаться с поставщиком!). В качестве эмпирического правила запомните: в обработчиках исключений обычно лучше придерживаться кон-

кретики, чем универсальности – идея, к которой мы еще вернемся при рассмотрении затруднений в следующей главе [1].

Что же тогда предпринять? Иерархии классов исключений полностью решают проблему. Вместо определения исключений вашей библиотеки как набора автономных классов организуйте их в виде дерева классов с общим суперклассом, охватывающим целую категорию:

```
# mathlib.py
class NumErr(Exception): pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass
def func():
    ...
    raise DivZero()
...и так далее...
```

В таком случае пользователям библиотеки просто нужно будет указывать общий суперкласс (т.е. категорию), чтобы перехватывать все исключения библиотеки и в текущий момент, и в будущем:

```
# client.py

import mathlib
try:
    mathlib.func(...)
except mathlib.NumErr:
    ...сообщить и выполнить восстановление...
```

Когда вы снова займетесь обновлением своего кода, то можете добавлять новые исключения как новые подклассы общего суперкласса:

```
# mathlib.py
...
class Uflow(NumErr): pass
```

В результате пользовательский код, который перехватывает исключения вашей библиотеки, останется работоспособным *без внесения изменений*. На самом деле впоследствии вы вольны добавлять, удалять и модифицировать исключения произвольным образом – до тех пор, пока клиенты указывают имя суперкласса, а он остается незатронутым, они защищены от изменения в вашем наборе исключений. Другими словами, исключения на основе классов обеспечивают лучший ответ на проблемы при сопровождении, чем могли бы строковые исключения.

Иерархии исключений, основанные на классах, также поддерживают предохранение состояния и наследование способами, которые делают их идеальными в более крупных программах. Тем не менее, чтобы понять их роли, сначала понадобится взглянуть, как классы исключений, определяемые пользователем, связаны со встроенными исключениями, от которых они унаследованы.

---

[1] Как предложил один из моих сообразительных студентов, модуль библиотеки мог бы также предоставлять объект кортежа, содержащий все исключения, которые потенциально способна генерировать библиотека – тогда клиент импортировал бы кортеж и указывал его в конструкции `except`, чтобы перехватывать все исключения библиотеки (вспомните, что наличие кортежа в `except` означает перехват *любого* из исключений). Когда позже добавляются новые исключения, библиотека просто расширяет экспортируемый кортеж. Такой прием сработал бы, но вам все равно пришлось бы поддерживать актуальность кортежа с генерируемыми исключениями внутри модуля библиотеки. Кроме того, иерархии классов предлагают больше преимуществ, нежели только категории – они также поддерживают наследуемое состояние и методы плюс модель настройки, чего индивидуальные исключения не делают.

Встроенные классы исключений

Вообще говоря, примеры из предыдущего раздела вовсе не были взяты с потолка. Все встроенные исключения, которые способен генерировать сам интерпретатор Python, представляют собой заранее определенные объекты классов. Вдобавок они организованы в неглубокую иерархию с универсальными категориями суперклассов и специфическими типами подклассов во многом подобно дереву классов исключений в предыдущем разделе.

В Python 3.X все хорошо известные исключения, которые вы видели (скажем, `SyntaxError`) в действительности являются лишь предварительно определенными классами, доступными в виде встроенных имен в модуле под названием `builtins`. В Python 2.X они находятся в `__builtin__` и также реализованы как атрибуты стандартного библиотечного модуля `exceptions`. В дополнение Python организует встроенные исключения в иерархию с целью поддержки разнообразных режимов перехвата. Ниже описаны примеры.

***`BaseException`*: самый верхний корень со стандартным методом вывода и конструктором**

Корневой суперкласс верхнего уровня для исключений. Этот класс не предназначен для прямого наследования классами, определяемыми пользователем (взамен используйте `Exception`). Он обеспечивает стандартное поведение вывода и предохранения состояния, наследуемое подклассами. Если вызвать встроенную функцию с экземпляром данного класса (например, через `print`), тогда он возвратит отображаемые строки аргументов конструктора, которые были переданы при создании экземпляра (или пустую строку в отсутствие аргументов). Вдобавок, если только подклассы не заместили конструктор класса `BaseException`, то все аргументы, переданные конструктору класса во время создания экземпляра, сохраняются в атрибуте `args` в форме кортежа.

***`Exception`*: корень определяемых пользователем исключений**

Корневой суперкласс верхнего уровня для исключений, связанных с приложением. Это непосредственный подкласс `BaseException` и суперкласс для каждого второго встроенного исключения кроме классов событий выхода в систему (`SystemExit`, `KeyboardInterrupt` и `GeneratorExit`). От него, а не от `BaseException`, должны наследоваться почти все классы исключений, определяемые пользователем. Когда такое соглашение соблюдается, то указание `Exception` в обработчике оператора `try` гарантирует, что программа будет перехватывать все кроме событий выхода в систему, которым обычно должно быть разрешено проходить. По существу `Exception` становится универсальной ловушкой в операторах `try` и обеспечивает более высокую точность, чем пустая конструкция `except`.

***`ArithmeticError`*: корень численных ошибок**

Подкласс `Exception` и суперкласс для всех численных ошибок. Его подклассы идентифицируют специфические численные ошибки: `OverflowError`, `ZeroDivisionError` и `FloatingPointError`.

***LookupError*: корень ошибок индексирования**

Подкласс Exception и суперкласс категории для ошибок индексирования в последовательностях и отображениях (IndexError и KeyError), а также некоторых ошибок при поиске Unicode.

И так далее — поскольку набор встроенных исключений подвержен частым изменениям, он полностью не документируется в книге. Дополнительные сведения ищите в справочниках вроде *Python Pocket Reference* (http://www.oreilly.com/catalog/9780596158088) или в руководстве по библиотекам Python. На самом деле дерево встроенных классов исключений в линейках Python 3.X и Python 2.X слегка отличается в аспектах, которые мы здесь опускаем, т.к. они не имеют отношения к примерам.

В Python 2.X вы также можете увидеть дерево встроенных классов исключений в справочном тексте по модулю exceptions (функция help обсуждалась в главах 4 и 15):

```
>>> import exceptions
>>> help(exceptions)
...справочный текст не показан...
```

Модуль был удален в Python 3.X, где актуальная справка доступна на других ресурсах.

Категории встроенных исключений

Дерево встроенных классов исключений позволяет выбирать, насколько конкретными или универсальными будут обработчики. Например, поскольку встроенное исключение ArithmeticError является суперклассом для более специфических исключений, таких как OverflowError и ZeroDivisionError:

- указывая в try исключение ArithmeticError, вы будете перехватывать возникшую численную ошибку любого вида;
- указывая в try исключение ZeroDivisionError, вы будете перехватывать только ошибку этого конкретного типа, но не другие.

Аналогично из-за того, что Exception в Python 3.X представляет собой суперкласс для всех исключений уровня приложения, в большинстве случаев вы можете применять его как *универсальную ловушку*. Эффект во многом похож на пустую конструкцию except, но исключениям выхода в систему разрешено проходить и распространятся надлежащим образом:

```
try:
    action()
except Exception:                # Исключения выхода в систему здесь
                                 #   не перехватываются
    ...обработать все исключения приложения...
else:
    ...обработать случай отсутствия исключений...
```

Однако в Python 2.X это работает не совсем универсально, потому что автономные определяемые пользователем исключения, реализованные в виде классических классов, не обязаны быть подклассами корневого класса Exception. Такая методика более надежна в Python 3.X, т.к. там требуется, чтобы все классы были производными от встроенных классов исключений. Тем не менее, как обсуждалось в предыдущей главе, даже в Python 3.X данная схема сопряжена с большинством тех же самых потенциальных ловушек, что и пустая конструкция except — она может перехватывать исключе-

ния, предназначенные для других мест, и способна маскировать подлинные ошибки в программе. Поскольку проблема настолько распространена, мы снова вернемся к ней при описании затруднений в следующей главе.

Независимо от того, будете вы задействовать категории в дереве встроенных классов исключений или нет, они служат хорошим примером; за счет использования похожих методик для исключений на основе классов в собственном коде вы можете предоставлять наборы исключений, отличающиеся гибкостью и легкостью модификации.

В версии *Python 3.3* были переделаны *иерархии встроенных исключений ввода-вывода и операционной системы*. Добавились новые специфические классы исключений, соответствующие распространенным номерам файловых и системных ошибок, которые вместе с остальными классами исключений, относящимися к вызовам операционной системы, собраны под суперклассом категории `OSError`. Первые из упомянутых имен исключений сохранены для обратной совместимости.

Ранее программы инспектировали данные, присоединенные к экземпляру исключения, чтобы выяснить, какая конкретно ошибка произошла, и возможно повторно генерировали остальные с целью дальнейшего распространения (модуль `errno` для удобства содержит имена, предварительно установленные в коды ошибок, а номер ошибки доступен в обобщенном кортеже как `V.args[0]` и в атрибуте `V.errno`):

```
c:\temp> py -3.2
>>> try:
...     f = open('nonesuch.txt')
... except IOError as V:
...     if V.errno == 2:                # Или errno.N, V.args[0]
...         print('No such file')       # Такой файл отсутствует
...     else:
...         raise                       # Распространить остальные
...
No such file
```

Код по-прежнему работает в Python 3.3, но с новыми классами программы на Python 3.3 и последующих версиях могут быть более конкретными в отношении исключений, которые они намерены обрабатывать, и игнорировать остальные:

```
c:\temp> py -3.3
>>> try:
...     f = open('nonesuch.txt')
... except FileNotFoundError:
...     print('No such file')           # Такой файл отсутствует
...
No such file
```

Более полное описание данного расширения и его классов ищите на других ресурсах.

Стандартный вывод и состояние

Встроенные исключения также обеспечивают вывод стандартного отображения и предохранение состояния, что зачастую и есть тем объемом логики, который требуется классам, определяемым пользователем. Если только вы не переопределили в своих

классах конструкторы, унаследованные от встроенных классов исключений, тогда любые передаваемые аргументы автоматически сохраняются в кортежном атрибуте args экземпляра и автоматически отображаются при выводе экземпляра. Пустые кортеж и отображаемая строка применяются, когда аргументы конструктора не передавались, а одиночный аргумент отображается сам по себе (не как кортеж).

Это объясняет, почему аргументы, переданные *встроенным* классам исключений, появляются в сообщениях об ошибках – любые аргументы конструктора присоединяются к экземпляру и отображаются при выводе экземпляра:

```
>>> raise IndexError           # То же, что и IndexError(): без аргументов
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 1, в <модуль>
Ошибка индекса

>>> raise IndexError('spam')  # Аргумент конструктора присоединяется и выводится
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: spam
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 1, в <модуль>
Ошибка индекса: spam
>> I = IndexError('spam')      # Аргумент доступен в атрибуте объекта
>>> I.args
('spam',)
>>> print(I)                   # Атрибуты отображаются при выводе вручную
spam
```

То же самое остается справедливым для *определяемых пользователем* исключений в Python 3.X (и для классов нового стиля в Python 2.X), потому что они наследуют методы конструктора и отображения, присутствующие в их встроенных суперклассах:

```
>>> class E(Exception): pass
...
>>> raise E
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.E
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 1, в <модуль>
__main__.E

>>> raise E('spam')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.E: spam
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 1, в <модуль>
__main__.E: spam

>>> I = E('spam')
>>> I.args
('spam',)
>>> print(I)
spam
```

В случае перехвата в операторе try объект экземпляра исключения предоставляет доступ к первоначальным аргументам конструктора и методу отображения:

```
>>> try:
...     raise E('spam')
... except E as X:
...     print(X)          # Отображает и сохраняет аргументы конструктора
...     print(X.args)
...     print(repr(X))
...
spam
('spam',)
E('spam',)
>>> try:        # Множество аргументов сохраняются/отображаются в виде кортежа
...     raise E('spam', 'eggs', 'ham')
... except E as X:
...     print('%s %s' % (X, X.args))
...
('spam', 'eggs', 'ham') ('spam', 'eggs', 'ham')
```

Обратите внимание, что объекты экземпляров исключений сами не являются строками, но используют протокол перегрузки операций `__str__`, исследованный в главе 30, чтобы предоставить отображаемые строки при выводе; для их конкатенации с настоящими строками выполняйте ручные преобразования: `str(X) + 'astr'`, `'%s' % X` и т.п.

Хотя такая автоматическая поддержка состояния и отображения полезна сама по себе, для удовлетворения специфических потребностей в отображении и предохранении состояния вы всегда можете переопределить такие унаследованные методы, как `__str__` и `__init__`, в подклассах `Exception` — чему и посвящен следующий раздел.

Специальное отображение при выводе

Как было показано в предыдущем разделе, по умолчанию экземпляры исключений, основанных на классах, отображают все, что было передано конструктору класса, когда они перехвачены и выводятся:

```
>>> class MyBad(Exception): pass
...
>>> try:
...     raise MyBad('Sorry--my mistake!')     # Сожалею--моя ошибка!
... except MyBad as X:
...     print(X)
...
Sorry--my mistake!
```

Такая унаследованная стандартная модель отображения также применяется, если исключение отображается как часть сообщения об ошибке, когда оно не перехвачено:

```
>>> raise MyBad('Sorry--my mistake!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyBad: Sorry--my mistake!

Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 1, в <модуль>
__main__.MyBad: Sorry--my mistake!
```

Для многих ролей этого достаточно. Но чтобы обеспечить более специализированное отображение, вы можете определить в своем классе один из двух методов перегрузки строкового представления (`__repr__` или `__str__`), возвращая строку, которую желательно отображать для исключения. Возвращаемая методом строка будет отображаться, если исключение либо перехвачено и выведено, либо достигло стандартного обработчика:

```
>>> class MyBad(Exception):
...     def __str__(self):
...         return 'Always look on the bright side of life...'
                  # Всегда смотри на светлую сторону жизни...
...
>>> try:
...     raise MyBad()
... except MyBad as X:
...     print(X)
...
Always look on the bright side of life...
>>> raise MyBad()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyBad: Always look on the bright side of life...
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 1, в <модуль>
__main__.MyBad: Always look on the bright side of life...
```

Все, что ваш метод возвращает, помещается в сообщения об ошибках для неперехваченных исключений и используется, когда исключения выводятся явно. В целях иллюстрации метод здесь возвращает жестко закодированную строку, но он может также выполнять произвольную текстовую обработку, скажем, применяя присоединенную к объекту экземпляра информацию состояния. Мы взглянем на варианты информации состояния в следующем разделе.

Здесь есть одна тонкость: обычно для целей отображения исключения вам придется переопределять метод `__str__`, поскольку встроенные суперклассы исключений уже его имеют, и в ряде контекстов `__str__` предпочтительнее `__repr__` — в том числе и при отображении сообщений об ошибках. Если вы определите метод `__repr__`, то вывод благополучно вызовет взамен метод `__str__` встроенного суперкласса!

```
>>> class E(Exception):
        def __repr__(self): return 'Not called!'    # Не вызывается!
>>> raise E('spam')
...
__main__.E: spam
>>> class E(Exception):
        def __str__(self): return 'Called!'         # Вызывается!
>>> raise E('spam')
...
__main__.E: Called!
```

Дополнительные сведения об этих специальных методах перегрузки операций ищите в главе 30.

Специальные данные и поведение

Помимо поддержки гибких иерархий классы исключений также предоставляют хранилище для добавочной информации состояния в форме атрибутов экземпляра. Как выяснилось ранее, встроенные суперклассы исключений предлагают стандартный конструктор, который автоматически сохраняет свои аргументы в кортежном атрибуте экземпляра по имени `args`. Несмотря на то что стандартный конструктор подходит во многих ситуациях, для более специализированных нужд мы можем предоставить собственный конструктор. В дополнение классы могут определять методы для использования в обработчиках, которые снабжают нас заранее реализованной логикой обработки исключений.

Предоставление деталей исключения

Когда исключение сгенерировано, оно способно пересекать границы произвольных файлов – оператор `raise`, генерирующий исключение, и перехватывающий его оператор `try` могут располагаться в совершенно разных файлах модулей. Хранить дополнительные детали в глобальных переменных обычно нереально, т.к. оператор может не знать, в каком файле находятся эти глобальные переменные. Передача добавочной информации состояния наряду с самим исключением позволяет оператору `try` получать доступ к ним более надежно.

Благодаря классам все достигается почти автоматически. Мы видели, что при генерации исключения Python передает объект экземпляра класса вместе с исключением. Код в операторах `try` способен обращаться к сгенерированному экземпляру, указывая дополнительную переменную после ключевого слова `as` в обработчике `except`, что обеспечивает естественную привязку для снабжения обработчика данными и поведением.

Например, программа разбора файлов данных может сигнализировать об ошибке формата путем генерации экземпляра исключения, который заполняется добавочными деталями об ошибке:

```
>>> class FormatError(Exception):
        def __init__(self, line, file):
            self.line = line
            self.file = file

>>> def parser():
        raise FormatError(42, file='spam.txt')     # Когда обнаружена ошибка

>>> try:
...     parser()
... except FormatError as X:
...     print('Error at: %s %s' % (X.file, X.line))
...
Error at: spam.txt 42
```

Здесь в конструкции `except` переменной `X` присваивается ссылка на экземпляр, который был создан, когда возникло исключение. Это дает доступ к атрибутам, присоединенным к экземпляру специальным конструктором. Хотя мы могли бы полагаться на стандартное предохранение состояния встроенными суперклассами, оно менее свойственно нашему приложению (и не поддерживает ключевые аргументы, применяемые в предыдущем примере):

```
>>> class FormatError(Exception): pass     # Унаследованный конструктор
>>> def parser():
        raise FormatError(42, 'spam.txt')  # Ключевые аргументы не разрешены!
>>> try:
...     parser()
... except FormatError as X:
...     print('Error at:', X.args[0], X.args[1])   # Не специфично для
                                                   #   данного приложения
...
Error at: 42 spam.txt
```

Предоставление методов исключений

Кроме разрешения специфичной для приложения информации состояния специальные конструкторы также лучше поддерживают добавочные линии поведения для объектов исключений. То есть в классе исключения могут также определяться *методы*, подлежащие вызову в обработчике. Скажем, в приведенном ниже коде из файла excparse.py добавлен метод, который использует информацию состояния исключения для автоматической регистрации ошибок в файле:

```
from __future__ import print_function    # Совместимость с Python 2.X

class FormatError(Exception):
   logfile = 'formaterror.txt'
   def __init__(self, line, file):
      self.line = line
      self.file = file
   def logerror(self):
      log = open(self.logfile, 'a')
      print('Error at:', self.file, self.line, file=log)
def parser():
   raise FormatError(40, 'spam.txt')
if __name__ == '__main__':
   try:
      parser()
   except FormatError as exc:
      exc.logerror()
```

После запуска сценарий записывает свои сообщения об ошибках в файл, реагируя на вызовы методов внутри обработчика исключений:

```
c:\code> del formaterror.txt
c:\code> py -3 excparse.py
c:\code> py -2 excparse.py
c:\code> type formaterror.txt
Error at: spam.txt 40
Error at: spam.txt 40
```

В таком классе методы (вроде logerror) могут также быть унаследованы из суперклассов, а атрибуты экземпляра (наподобие line и file) предоставляют место для сохранения информации состояния, которая предлагает дополнительное содержимое для применения в более поздних вызовах методов. Кроме того, классы исключений вольны настраивать и расширять унаследованное поведение:

```
class CustomFormatError(FormatError):
   def logerror(self):
      ...что-то уникальное...
raise CustomFormatError(...)
```

Другими словами, поскольку они определены с помощью классов, все преимущества ООП, которые обсуждались в части Part VI, доступны для использования с исключениями в Python.

Два финальных замечания: во-первых, сгенерированный объект экземпляра, присвоенный `exc` в этом коде, также доступен обобщенно как второй элемент в результирующем кортеже вызова `sys.exc_info()` — инструмента, который возвращает информацию о самом недавнем исключении. Такой интерфейс должен применяться, если вы не указываете имя исключения в конструкции `except`, но все равно нуждаетесь в доступе к возникшему исключению или присоединенной информации состояния либо методам. Во-вторых, хотя метод `logerror` нашего класса добавляет специальное сообщение в журнальный файл, он мог бы также генерировать стандартное сообщение об ошибке Python с трассировкой стека, используя инструменты из стандартного библиотечного модуля `traceback`, который работает с объектами трассировки.

Рассмотрение `sys.exc_info` и объектов трассировки продолжается в следующей главе.

Резюме

В главе демонстрировалась методика создания определяемых пользователем исключений. Вы узнали, что начиная с версий Python 2.6 and 3.0, исключения реализуются в виде объектов экземпляров классов (предшествующая альтернативная модель исключений на основе строк была доступна в более ранних версиях, но теперь объявлена устаревшей). Классы исключений поддерживают концепцию иерархий исключений, которая облегчает сопровождение, позволяет присоединять к исключениям данные и поведение как атрибуты и методы экземпляров, а также разрешает исключениям наследовать данные и поведение от суперклассов.

Как выяснилось, указание суперкласса в операторе `try` обеспечивает перехват этого класса и всех его подклассов ниже в дереве классов. Суперклассы становятся названиями категорий исключений, а подклассы — более конкретными типами исключений внутри таких категорий. Также было показано, что встроенные суперклассы исключений, от которых должны наследоваться подклассы, предоставляют удобные стандартные методы для вывода и предохранения состояния, в случае необходимости допускающие переопределение.

Следующая глава завершает очередную часть книги исследованием ряда сценариев применения исключений и обзором инструментов, часто используемых программистами на Python. Но прежде чем переходить к ее чтению, ответьте на контрольные вопросы главы, чтобы закрепить полученные знания.

Проверьте свои знания: контрольные вопросы

1. Какие два новых ограничения накладываются на определяемые пользователем исключения в Python 3.X?
2. Как сгенерированные исключения, основанные на классах, сопоставляются с обработчиками?
3. Назовите два способа присоединения информации контекста к объектам исключений.
4. Назовите два способа указания текста сообщений об ошибках для объектов исключений.
5. Почему исключения на основе строк в наше время больше не должны использоваться?

Проверьте свои знания: ответы

1. В Python 3.X исключения должны определяться посредством классов (т.е. генерируется и перехватывается объект экземпляра класса). Вдобавок классы исключений обязаны быть производными от встроенного класса `BaseException`; в большинстве программ они наследуются от его подкласса `Exception`, чтобы поддерживать универсальные обработчики для нормальных разновидностей исключений.

2. Исключения на основе классов сопоставляются по отношениям с суперклассами: указание суперкласса в обработчике исключений приводит к перехвату экземпляров данного класса и экземпляров всех его подклассов ниже в дереве классов. По этой причине вы можете думать о суперклассах как об общих категориях исключений, а о подклассах как о более специфичных типах исключений внутри таких категорий.

3. Вы можете присоединять информацию контекста к основанным на классах исключениям путем заполнения атрибутов экземпляра в сгенерированном объекте экземпляра обычно внутри специального конструктора класса. Для более простых потребностей встроенные суперклассы исключений предлагают конструктор, который автоматически сохраняет свои аргументы в экземпляре (в форме кортежа в атрибуте `args`). В обработчиках исключений вы указываете переменную, которой должен присваиваться сгенерированный экземпляр, после чего с помощью этого имени получаете доступ к присоединенной информации состояния и вызываете любые методы, определенные в классе.

4. Текст сообщений об ошибках в исключениях на основе классов может указываться посредством специального метода перегрузки операций `__str__`. Для более простых потребностей встроенные суперклассы исключений автоматически отображают все, что передается конструкторам классов. Операции вроде `print` и `str` автоматически извлекают отображаемую строку объекта исключения, когда он выводится либо явно, либо как часть сообщения об ошибке.

5. Потому что так сказал Гвидо — они были удалены, начиная с версий Python 2.6 и 3.0. Вероятно, на то имелись веские причины: исключения на основе строк не поддерживали категории, информацию состояния и наследование поведения, как это делают исключения, основанные на классах. По существу исключения на основе строк было легче использовать на первых порах, пока программы имели небольшие размеры, но по мере роста программ применять такие исключения становилось все труднее.

6. С требованием того, чтобы исключения были классами, связаны также недостатки: *нарушение работы* существующего кода и создание заблаговременной *зависимости от знаний* — новички обязаны сначала изучить классы и ООП, прежде чем они сумеют реализовывать новые исключения или даже по-настоящему понимать исключения вообще. Фактически именно потому эта относительно прямолинейная тема была отложена вплоть до настоящего места книги. Так или иначе, зависимости подобного рода — далеко не редкость в современном языке Python.

ГЛАВА 36

Проектирование с использованием исключений

Настоящая глава завершает эту часть книги совокупностью тем, касающихся проектирования с применением исключений, и набором распространенных сценариев использования, после чего обсуждаются затруднения и предлагаются упражнения. Поскольку глава заканчивает порцию книги, в целом посвященную фундаментальным основам, в ней также приводится краткий обзор инструментов для разработки, чтобы помочь вам перейти из разряда новичков в языке Python в разработчики прикладных приложений на Python.

Вложение обработчиков исключений

До сих пор в большинстве примеров для перехвата исключений применялся только одиночный оператор `try`, но что произойдет, если один `try` физически вложить внутрь другого `try`? Вдобавок что означает, когда в `try` вызывается функция, которая запускает еще один `try`? Формально операторы `try` можно вкладывать с точки зрения как синтаксиса, так и потока управления на протяжении выполнения кода. Ранее я кратко об этом упоминал, но давайте проясним саму идею.

Оба сценария можно понять, если осознавать, что во время выполнения интерпретатор Python укладывает операторы `try` *в стопку*. При генерации исключения Python возвращается к самому последнему введенному оператору `try` с дающей совпадение конструкцией `except`. Из-за того, что каждый оператор `try` оставляет маркер, интерпретатор Python может переходить обратно на более ранние операторы `try` за счет инспектирования уложенных в стопку маркеров. Именно такое вложение активных обработчиков подразумевается, когда речь идет о распространении исключений вверх до "более высоких" обработчиков — обработчики подобного рода являются просто операторами `try`, введенными *раньше* в потоке выполнения программы.

На рис. 36.1 показано, что происходит при вложении операторов `try` с конструкциями `except` во время выполнения. Объем кода в блоке `try` может быть значительным, и он способен содержать вызовы функций, которые обращаются к другому коду, отслеживающему те же самые исключения. Когда в конечном итоге генерируется исключение, Python переходит обратно на самый последний введенный оператор `try`, в котором указано сгенерированное исключение, выполняет конструкцию `except` данного оператора и затем возобновляет выполнение после этого `try`.

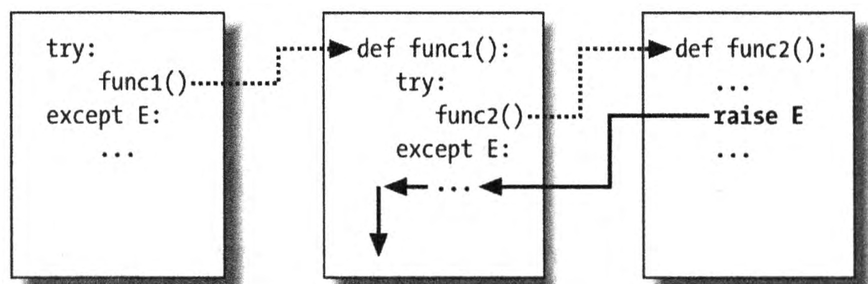


Рис. 36.1. *Вложенные операторы* `try/except`*: когда генерируется исключение (вами либо интерпретатором Python), управление передается самому последнему введенному оператору* `try` *с конструкцией* `except`*, дающей совпадение, и программа возобновляет выполнение после этого оператора* `try`*. Конструкции* `except` *перехватывают и прекращают распространение исключения — именно в них производится обработка и восстановление после исключений*

После того, как исключение перехвачено, время его существования заканчивается — управление не передается *всем* подходящим операторам `try`, в которых указано имя исключения; шанс его обработать предоставляется только первому (т.е. самому недавнему) `try`. Например, на рис. 36.1 оператор `raise` в функции `func2` возвращает управление обработчику в `func1` и затем программа продолжает выполнение внутри `func1`.

Напротив, когда операторы `try`, содержащие только конструкции `finally`, оказываются вложенными, то при возникновении исключения *каждый* блок `finally` выполняется по очереди — интерпретатор Python продолжает распространение исключения выше до других операторов `try` и со временем возможно до стандартного обработчика верхнего уровня (выводящего стандартное сообщение об ошибке). Как иллюстрируется на рис. 36.2, конструкции `finally` не уничтожают исключение — они просто указывают код, подлежащий выполнению при выходе из каждого `try` в течение процесса распространения исключений. Если в момент генерации исключения есть много активных конструкций `try/finally`, тогда *все* они будут выполнены при условии, что исключение где-то по пути не перехватывается каким-то оператором `try/except`.

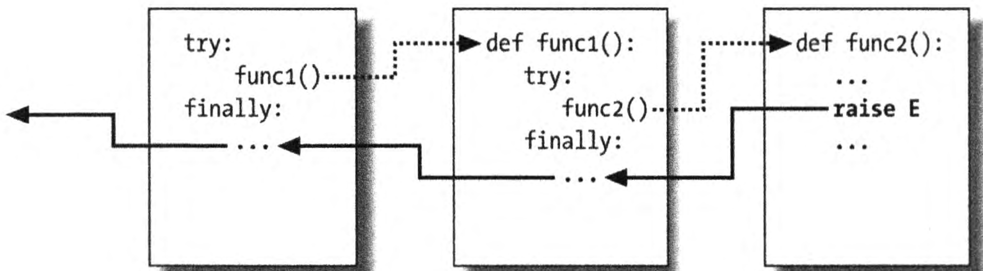


Рис. 36.2. *Вложенные операторы* `try/finally`*: когда здесь генерируется исключение, управление передается самому последнему введенному оператору* `try` *для выполнения его конструкции* `finally`*, но затем исключение продолжает распространяться до всех конструкций* `finally` *во всех активных операторах* `try` *и со временем достигнет стандартного обработчика верхнего уровня, выводящего сообщение об ошибке. Конструкции* `finally` *перехватывают (но не прекращают) исключение — они предназначены для указания действий, выполняемых "при выходе"*

Другими словами, то, куда управление в программе переходит в случае генерации исключения, зависит целиком от того, *где оно находилось* — это работа потока управления во время выполнения сценария, а не только синтаксиса. По существу распространение исключения продолжается обратно сквозь время до операторов try, вход в которые происходил, но выход пока нет. Такое распространение останавливается, как только поток управления раскручивается до дающей совпадение конструкции except, но не так, как оно проходит через конструкции finally на своем пути.

Пример: вложение в потоке управления

Чтобы прояснить эту концепцию вложения, давайте рассмотрим пример. В файле модуля nestexc.py, код которого представлен ниже, определяются две функции. Функция action2 реализована для генерации исключения (складывать числа и последовательности нельзя), а функция action1 содержит внутри вызов action2 в обработчике try, чтобы перехватывать исключение:

```
def action2():
    print(1 + [])                # Генерирует TypeError

def action1():
    try:
        action2()
    except TypeError:            # Самый последний совпадающий try
        print('inner try')       # внутренний оператор try

try:
    action1()
except TypeError:                # Здесь только в случае повторной
                                 #   генерации исключения в action1
    print('outer try')           # внешний оператор try

% python nestexc.py
inner try
```

Однако обратите внимание, что код верхнего уровня модуля в конце файла тоже помещает вызов action1 внутрь обработчика try. Когда action2 сгенерирует исключение TypeError, появятся два активных оператора try — один в action1 и один на верхнем уровне файла модуля. Интерпретатор Python выберет и выполнит самый недавний try с совпадающей конструкцией except, которым в данном случае будет try внутри action1.

Опять-таки место, куда попадает исключение, зависит от того, как поток управления проходит через программу во время выполнения. По указанной причине, чтобы узнать, где вы окажетесь, нужно знать, где вы были. В приведенной ситуации место обработки исключения определяется в большей степени потоком управления, нежели синтаксисом операторов. Тем не менее, мы можем вкладывать обработчики исключений также и синтаксически — эквивалентный сценарий описан в следующем разделе.

Пример: синтаксическое вложение

Как упоминалось при рассмотрении нового объединенного оператора try/except/finally в главе 34, операторы try допускается вкладывать синтаксически за счет их позиции в исходном коде:

```
try:
    try:
        action2()
```

```
    except TypeError:          # Самый последний совпадающий try
        print('inner try')
except TypeError:              # Здесь только в случае повторной генерации
    print('outer try')         # исключения во вложенном обработчике
```

Однако на самом деле в коде всего лишь настраивается та же самая структура вложения обработчиков, что и в предыдущем примере (и ведущая себя идентично). В действительности синтаксическое вложение работает в точности как случаи, изображенные на рис. 36.1 и 36.2. Единственная разница в том, что вложенные обработчики физически внедрены в блок `try`, а не реализованы где-то в функциях, которые вызываются из блока `try`. Скажем, при исключении запускаются все вложенные обработчики `finally`, вложены они синтаксически или посредством прохождения потока управления через физически разделенные части кода:

```
>>> try:
...     try:
...         raise IndexError
...     finally:
...         print('spam')
... finally:
...     print('SPAM')
...
spam
SPAM
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError
Трассировка (самый последний вызов указан последним):
  Файл "<stdin>", строка 3, в <модуль>
Ошибка индекса
```

Графическая иллюстрация работы такого кода приводилась на рис. 36.2; результат тот же, но логика функции здесь была записана как вложенные операторы. Более полезный пример синтаксического вложения можно найти в файле `except-finally.py`:

```
def raise1(): raise IndexError
def noraise(): return
def raise2(): raise SyntaxError

for func in (raise1, noraise, raise2):
    print('<%s>' % func.__name__)
    try:
        try:
            func()
        except IndexError:
            print('caught IndexError')
    finally:
        print('finally run')
    print('...')
```

Код перехватывает исключение, если оно сгенерировано, и выполняет действие завершения `finally` независимо от того, возникало ли исключение. Его понимание может потребовать какого-то времени, но эффект будет таким же, как при объединении конструкций `except` и `finally` в единственном операторе `try` в Python 2.5 и последующих версиях:

```
% python except-finally.py
<raise1>
caught IndexError
finally run
...
<noraise>
finally run
...
<raise2>
finally run
Traceback (most recent call last):
  File "except-finally.py", line 9, in <module>
    func()
  File "except-finally.py", line 3, in raise2
    def raise2(): raise SyntaxError
SyntaxError: None
Трассировка (самый последний вызов указан последним):
  Файл "except-finally.py", строка 9, в <модуль>
    func()
  Файл "except-finally.py", строка 3, в <модуль>
    def raise2(): raise SyntaxError
Ошибка индекса: None
```

Как было показано в главе 34, начиная с версии Python 2.5, конструкции `except` и `finally` можно смешивать в одном операторе `try`. Наряду с поддержкой множества конструкций `except` это делает часть описанного выше синтаксического вложения ненужным, хотя эквивалентное вложение времени выполнения распространено в более крупных программах на Python. Кроме того, синтаксическое вложение по-прежнему работает в наши дни, все еще может встречаться в коде, написанном до выхода версии Python 2.5, способно делать разрозненные роли `except` и `finally` более явными и может использоваться в качестве методики для реализации альтернативных линий поведения при обработке исключений.

Идиомы исключений

Итак, мы ознакомились с механизмом, лежащим в основе исключений. Теперь давайте рассмотрим ряд других способов его типичного применения.

Прерывание множества вложенных циклов: "безусловный переход"

Как упоминалось в начале этой части книги, исключения часто могут использоваться для исполнения тех же ролей, что и операторы "безусловного перехода" в других языках, чтобы реализовывать более произвольные передачи управления. Тем не менее, исключения предоставляют структурированный способ, который локализует переход к специфическому блоку вложенного кода.

В данной роли оператор `raise` похож на "безусловный переход", а конструкции `except` и имена исключений занимают место программных меток. Таким способом можно выходить только из кода, помещенного внутрь `try`, но возможность очень важна – по-настоящему произвольные операторы "безусловного перехода" способны сделать код чрезвычайно сложным для понимания и сопровождения.

Например, оператор break языка Python выходит только из одиночного ближайшего объемлющего цикла, но мы всегда можем применить исключения, чтобы в случае необходимости прервать более одного вложенного цикла:

```
>>> class Exitloop(Exception): pass
...
>>> try:
...     while True:
...         while True:
...             for i in range(10):
...                 if i > 3: raise Exitloop    # break выходит только
                                                #   из одного уровня вложенности
...                 print('loop3: %s' % i)
...             print('loop2')
...         print('loop1')
... except Exitloop:
...     print('continuing')         # Или просто pass, чтобы двигаться дальше
...
loop3: 0
loop3: 1
loop3: 2
loop3: 3
continuing
>>> i
4
```

Если вы измените raise на break, то получите бесконечный цикл, т.к. произойдет выход только из наиболее глубоко вложенного цикла for и попадание в цикл второго уровня вложенности. Код затем выведет loop2 и начнет цикл for заново.

Вдобавок обратите внимание, что переменная i остается такой же, какой она была после выхода из оператора try. Присваивания переменных, выполненные внутри try, в общем случае не отменяются, хотя переменные экземпляров исключений, указанные в заголовках конструкций except, локализуются в пределах этих конструкций, а локальные переменные любых функций, которые завершились из-за raise, отбрасываются. Формально локальные переменные активных функций выталкиваются из стека вызовов и в результате объекты, на которые они ссылаются, могут быть подвергнуты сборке мусора, но такой шаг выполняется автоматически.

Исключения не всегда являются ошибками

В Python все ошибки являются исключениями, но не все исключения — ошибками. Скажем, в главе 9 было показано, что методы чтения файловых объектов по достижении конца файла возвращают пустую строку. Напротив, встроенная функция input (которую мы впервые встретили в главе 3, задействовали в интерактивном цикле в главе 10 и узнали, что в Python 2.X она называется raw_input) при каждом вызове читает строку текста из стандартного входного потока sys.stdin и при достижении конца файла генерирует встроенное исключение EOFError.

В отличие от файловых методов функция input не возвращает пустую строку — пустая строка из input означает пустую строку в файле. Однако, несмотря на свое имя, исключение EOFError в данном контексте представляет собой просто сигнал, не ошибку. По причине такого поведения, если признак конца файла не должен завершать работу сценария, то вызов input часто встречается помещенным внутрь обработчика try и вложенным в цикл, как в следующем коде:

```
while True:
    try:
        line = input()      # Чтение строки из stdin (raw_input в Python 2.X)
    except EOFError:
        break               # Выход из цикла по достижении конца файла
    else:
        ...обработка очередной строки...
```

Несколько других встроенных исключений аналогичным образом являются сигналами, а не ошибками – например, вызов `sys.exit()` и нажатие <Ctrl+C> на клавиатуре генерируют соответственно исключения `SystemExit` и `KeyboardInterrupt`.

В Python также имеется набор встроенных исключений, которые вместо ошибок представляют *предупреждения*; ряд из них служит для того, чтобы сигнализировать об использовании устаревших (постепенно исчезающих) языковых средств. За дополнительной информацией о встроенных исключениях обращайтесь в руководство по стандартной библиотеке, а сведения об исключениях, генерируемых как предупреждения, ищите в документации по модулю `warnings`.

Функции могут сигнализировать об условиях с помощью `raise`

Определяемые пользователем исключения способны также сигнализировать о неошибочных условиях. Скажем, процедура поиска может быть реализована так, чтобы в случае нахождения совпадения генерировать исключение, а не возвращать флаг состояния для интерпретации в вызывающем коде. Ниже обработчик исключения выполняет работу оператора `if/else` по проверке возвращаемого значения:

```
class Found(Exception): pass

def searcher():
    if ...успех...:
        raise Found()       # Генерировать исключения вместо возвращения флагов
    else:
        return

try:
    searcher()
except Found:               # Исключение, если элемент был найден
    ...успех...
else:                       # иначе элемент не был найден
    ...неудача...
```

В более общем смысле такая кодовая структура также может быть полезной для любой функции, которая не способна возвращать *индикаторное значение*, обозначающее успех или неудачу. Например, если в широко применяемой функции все объекты являются допустимыми возвращаемыми значениями, то сигнализировать об условии неудачи с помощью любого возвращаемого значения попросту невозможно. Исключения предлагают способ предупреждения о результатах без возвращаемого значения:

```
class Failure(Exception): pass

def searcher():
    if ...успех...:
        return ...найденный элемент...
    else:
        raise Failure()

try:
```

```
    item = searcher()
except Failure:
    ...элемент не найден...
else:
    ...использовать элемент...
```

Поскольку язык Python глубоко динамически типизированный и полиморфный, исключения в отличие от индикаторных возвращаемых значений являются предпочитаемым способом сигнализации о таких условиях.

Закрытие файлов и серверных подключений

Примеры такой категории встречались в главе 34. Тем не менее, в качестве резюме отметим, что инструменты обработки исключений также часто используются для обеспечения финализации системных ресурсов независимо от того, возникала ошибка во время обработки или нет.

Скажем, некоторые серверы требуют закрытия подключений для завершения сеанса. Аналогично выходные файлы могут требовать закрытия, чтобы сбросить свои буферы на диск для ожидающих клиентов, а входные файлы могут потреблять файловые ресурсы, не будучи закрытыми. Хотя при сборке мусора файловые объекты автоматически закрываются, если все еще открыты, в некоторых реализациях Python иногда нелегко выяснить, когда сборка мусора произойдет.

В главе 34 было показано, что наиболее универсальный и явный способ гарантировать выполнение завершающих действий для индивидуального блока кода предусматривает применение оператора `try/finally`:

```
myfile = open(r'C:\code\textdata', 'w')
try:
    ...обработка myfile...
finally:
    myfile.close()
```

Также выяснилось, что в Python 2.6, 3.0 и последующих версиях некоторые объекты потенциально облегчают выполнение завершающих действий, предоставляя *диспетчеры контекстов*, которые автоматически завершают работу или закрывают объекты в случае использования с оператором `with/as`:

```
with open(r'C:\code\textdata', 'w') as myfile:
    ...обработка myfile...
```

Так какой вариант лучше? Как обычно, все зависит от ваших программ. По сравнению с традиционным оператором `try/finally` диспетчеры контекстов *менее явные*, что противоречит общей философии проектирования в Python. Диспетчеры контекстов также вероятно *менее универсальны* – они доступны только для избранных объектов, а реализация определяемых пользователем диспетчеров контекстов для учета общих требований к завершению сложнее написания операторов `try/ finally`.

С другой стороны, применение диспетчеров контекстов требует *меньше кода*, чем использование `try/finally`, как демонстрировалось в предшествующих примерах. Вдобавок к действиям выхода протокол диспетчеров контекстов поддерживает действия *входа*. В действительности они могут сберечь строку кода, когда никакие исключения не ожидаются (хотя за счет дальнейшего вложения и добавления отступов к логике обработки файлов):

```
myfile = open(filename, 'w')      # Традиционная форма
...обработка myfile...
```

```
myfile.close()

with open(filename) as myfile:    # Форма с диспетчером контекста
    ...обработка myfile...
```

Однако неявная обработка исключений оператора with делает его более сопоставимым с явной обработкой исключений try/finally. Несмотря на то что try/finally является шире применимой методикой, диспетчеры контекстов могут оказаться предпочтительнее, где они уже доступны либо их добавочная сложность оправдана.

Отладка с помощью внешних операторов try

Вы также можете использовать обработчики исключений для замещения стандартного поведения обработки исключений на верхнем уровне Python. Поместив целую программу (или обращение к ней) во внешний оператор try в своем коде верхнего уровня, вы можете перехватывать любое исключение, которое произошло во время выполнения программы, посредством чего отменяя стандартное завершение работы программы.

Ниже пустая конструкция except перехватывает любое неперехваченное исключение, сгенерированное во время выполнения программы. Чтобы получить действительное исключение, которое произошло в таком режиме, извлеките результат вызова функции sys.exc_info из встроенного модуля sys; она возвращает кортеж, первый элемент которого содержит класс текущего исключения и сгенерированный объект экземпляра (вскоре функция sys.exc_info будет описана более детально):

```
try:
    ...запуск программы...
except:                          # Сюда поступают все неперехваченные исключения
    import sys
    print('uncaught!', sys.exc_info()[0], sys.exc_info()[1])
```

Такая структура обычно применяется на этапе разработки для сохранения программ в активном состоянии даже после возникновения ошибок — внутри цикла она позволяет прогонять дополнительные тесты без необходимости в перезапуске. Прием также используется при тестировании кода других программ, как объясняется в следующем разделе.

В качестве стороннего замечания: чтобы узнать больше об обработке завершения программ *без* восстановления после них, ознакомьтесь также со стандартным библиотечным модулем Python под названием atexit. Настроить работу обработчика исключений верхнего уровня возможно и с помощью функции sys.excepthook. Эти и другие связанные инструменты описаны в руководстве по библиотеке Python.

Выполнение внутрипроцессных тестов

Ряд только что рассмотренных кодовых схем можно комбинировать в испытательном приложении, которое тестирует другой код внутри того же процесса. Следующий неполный код демонстрирует общую модель:

```
import sys
log = open('testlog', 'a')
from testapi import moreTests, runNextTest, testName
def testdriver():
    while moreTests():
```

```
        try:
            runNextTest()
        except:
            print('FAILED', testName(), sys.exc_info()[:2], file=log)
        else:
            print('PASSED', testName(), file=log)
testdriver()
```

Функция testdriver выполняет в цикле серию обращений к тестам (в примере мы абстрагируемся от модуля testapi). Поскольку неперехваченное исключение в тестовом сценарии нормально прекратило бы работу испытательного приложения, вам необходимо поместить обращения к тестовым сценариям внутрь оператора try, если вы хотите продолжать процесс тестирования после неудачного прохождения теста. Пустая конструкция except обычным образом перехватывает любые неперехваченные исключения, сгенерированные тестовым сценарием, и применяет sys.exc_info для регистрации исключения в журнальном файле. Конструкция else выполняется, когда никаких исключений не было — успешное прохождение теста.

Такой стереотипный код типичен для систем, которые тестируют функции, модули и классы, запуская их в *том же самом процессе*, где выполняется испытательное приложение. Тем не менее, на практике тестирование может оказаться гораздо более сложным, чем проиллюстрированное выше. Например, для тестирования *внешних программ* вы могли бы взамен проверять коды состояния или вывод, порожденный такими инструментами запуска программ, как os.system и os.popen, которые использовались ранее в книге и раскрываются в руководстве по стандартной библиотеке. Инструменты подобного рода обычно не генерируют исключения для ошибок во внешних программах — на самом деле тестовые сценарии могут выполняться параллельно с испытательным приложением.

В конце главы будут кратко представлены более полные фреймворки тестирования, входящие в состав Python, вроде doctest и PyUnit, которые предлагают инструменты для сравнения ожидаемого вывода с фактическими результатами.

Дополнительные сведения о функции sys.exc_info

Результат вызова sys.exc_info, который применялся в предшествующих двух разделах, дает возможность обработчику исключений получать доступ к самому последнему сгенерированному исключению обобщенным образом. Это особенно полезно при использовании пустой конструкции except для слепого перехвата всех исключений, чтобы выяснить, какое исключение было сгенерировано:

```
try:
    ...
except:
    # sys.exc_info()[0:2] представляет класс и экземпляр исключения
```

Если никакие исключения не обрабатывались, тогда такой вызов возвращает кортеж с тремя значениями None. В противном случае значениями будут (*type*, *value*, *traceback*):

- type — класс обрабатываемого исключения;
- value — экземпляр класса исключения, который был сгенерирован;
- traceback — объект трассировки, который представляет стек вызовов в месте, где первоначально возникло исключение, и применяется модулем traceback для генерации сообщений об ошибках.

Мы видели в предыдущей главе, что функция `sys.exc_info` иногда может быть полезной для установления конкретного типа исключения при перехвате суперклассов категорий исключений. Однако поскольку в такой ситуации получить тип исключения можно также за счет извлечения атрибута `__class__` экземпляра, полученного через `as`, вызов `sys.exc_info` часто избыточен кроме случая с пустой конструкцией `except`:

```
try:
    ...
except General as instance:
    # instance.__class__ представляет класс исключения
```

Как уже было показано, использование здесь `Exception` для имени исключения *General* аналогично пустой конструкции `except` будет перехватывать все исключения, отличающиеся от событий выхода в систему, но менее экстремально, и по-прежнему предоставлять доступ к экземпляру исключения и его классу. Тем не менее, применение интерфейсов объекта экземпляра и *полиморфизма* зачастую является более эффективным подходом, чем проверка типов исключений — методы исключений могут определяться для каждого класса и выполняться обобщенным образом:

```
try:
    ...
except General as instance:
    # instance.method() делает то, что нужно, для данного экземпляра
```

Как обычно, слишком большая конкретика в Python способна ограничить гибкость вашего кода. Полиморфный подход наподобие того, что использовался в последнем примере, в целом лучше поддерживает будущее развитие, нежели явные проверки и действия, специфичные к типам.

Отображение сообщений об ошибках и трассировок

Наконец, объект трассировки исключения, доступный в результате вызова `sys.exc_info` из предыдущего раздела, также применяется стандартным библиотечным модулем `traceback` для формирования стандартного сообщения об ошибке и отображения стека вручную. Файл модуля `traceback` содержит несколько интерфейсов, поддерживающих обширную настройку, раскрыть полностью которые здесь невозможно из-за нехватки места, но сами основы просты. Взгляните на содержимое подходящим образом названного файла, `badly.py`:

```
import traceback

def inverse(x):
    return 1 / x

try:
    inverse(0)
except Exception:
    traceback.print_exc(file=open('badly.exc', 'w'))
print('Bye')
```

В коде используется удобная функция `print_exc` из модуля `traceback`, которая по умолчанию потребляет данные `sys.exc_info`; после запуска сценарий выводит сообщение об ошибке в файл, что удобно в тестовых программах, которым необходимо перехватывать ошибки, но вдобавок полноценно их регистрировать:

```
c:\code> python badly.py
Bye

c:\code> type badly.exc
Traceback (most recent call last):
  File "C:\Code\badly.py", line 7, in <module>
    inverse(0)
  File "C:\Code\badly.py", line 4, in inverse
    return 1 / x
ZeroDivisionError: division by zero
Трассировка (самый последний вызов указан последним):
  Файл "C:\Code\badly.py", строка 7, в <модуль>
    inverse(0)
  Файл "C:\Code\badly.py", строка 4, в inverse
    return 1 / x
Ошибка деления на ноль: деление на ноль
```

Более подробные сведения об объектах трассировки, модуле `traceback`, в котором они применяются, и связанных темах ищите в других справочных ресурсах и руководствах.

Примечание, касающееся нестыковки версий. Более старые инструменты `sys.exc_type` и `sys.exc_value` по-прежнему работают в Python 2.X для извлечения типа и значения самого последнего исключения, но они могут управлять только одним глобальным исключением для целого процесса. В Python 3.X упомянутые два имени были удалены. Более новый и предпочтительный вызов `sys.exc_info()`, доступный в Python 2.X и Python 3.X, взамен отслеживает информацию об исключении каждого потока и потому специфичен к потокам. Разумеется, такое отличие важно только в случае использования множества потоков в программах на Python (тема, выходящая за рамки настоящей книги), но в Python 3.X проблема усиливается. За дополнительными сведениями обращайтесь на другие ресурсы.

Советы по проектированию с использованием исключений и связанные с ними затруднения

В этой главе я собрал вместе советы по проектированию и затруднения, т.к. выясняется, что наиболее распространенные затруднения в значительной степени уходят своими корнями в проблемы, касающиеся проектирования. В общем и целом применять исключения в Python несложно. Подлинное искусство связано с принятием решения о том, до какой степени конкретными или универсальными должны быть конструкции `except`, и сколько кода подлежит помещению внутрь операторов `try`. Давайте займемся сначала вторым вопросом.

Что должно быть помещено внутрь операторов `try`?

В принципе вы могли бы поместить каждый оператор в своем сценарии внутрь собственного оператора `try`, но поступать так попросту нелепо (тогда и операторы `try` пришлось бы помещать внутрь операторов `try`!). То, что помещается внутрь `try`, в действительности представляет собой задачу проектирования, которая выходит за рамки самого языка, и станет более очевидным в ходе работы. А пока ниже приведено несколько эмпирических правил.

- Операции, которые обычно терпят неудачу, как правило, должны помещаться внутрь операторов try. Например, операции, взаимодействующие с системным состоянием (открытия файлов, обращения к сокетам и т.п.) будут главными кандидатами для try.
- Однако существуют отклонения от предыдущего правила — в простом сценарии у вас может возникнуть желание, чтобы неудачи таких операций уничтожали вашу программу, а не были перехвачены и проигнорированы. Сказанное особенно справедливо, если неудача является устойчивой ошибкой. Неудачи в Python обычно приводят к выводу полезных сообщений об ошибках (не к полным отказам) и являются наилучшим исходом, на который могли бы рассчитывать некоторые программы.
- Действия по завершении должны быть реализованы в операторах try/finally, чтобы гарантировать их выполнение, если только не оказывается доступным диспетчер контекста как вариант with/as. Форма оператора try/ finally позволяет запускать код независимо от того, возникали исключения в произвольных сценариях или нет.
- Иногда удобнее помещать вызов крупной функции внутрь единственного оператора try, не засоряя саму функцию множеством операторов try. В итоге все исключения из функции проникают в try вокруг вызова, а объем кода внутри функции сокращается.

Реализуемые типы программ, вероятно, также будут оказывать влияние на объем кода обработки исключений. Например, *серверы* обычно обязаны функционировать постоянно и потому, скорее всего, потребуют наличия операторов try для перехвата и восстановления после исключений. Программы внутрипроцессного *тестирования* показанного в главе вида возможно тоже будут обрабатывать исключения. Тем не менее, простые одноразовые сценарии часто полностью игнорируют обработку исключений, поскольку неудача на любом шаге требует прекращения работы сценария.

Перехват слишком многого: избегайте использования пустой конструкции except и конструкции except Exception

Как уже упоминалось, универсальность обработчиков исключений — определяющий проектный выбор. Python позволяет быть разборчивым в том, какие исключения перехватывать, но иногда необходимо соблюдать осторожность, чтобы обработчики не становились излишне инклюзивными. Например, вы видели, что пустая конструкция except перехватывает *любое* исключение, которое может сгенерироваться на стадии выполнения кода в блоке try.

Писать такой код легко и временами желательно, но вы можете также в итоге перехватить ошибку, которая ожидается обработчиком try выше в структуре вложения исключений. Скажем, обработчик исключений вроде следующего перехватывает и прекращает *любое* исключение, которое его достигает, невзирая на то, что оно ожидается еще одним обработчиком:

```
def func():
    try:
        ...             # Здесь генерируется исключение IndexError
    except:
        ...             # Но все поступает сюда и исчезает!
```

```
try:
    func()
except IndexError:    # Исключение IndexError должно обрабатываться здесь
    ...
```

Вероятно хуже то, что такой код может также перехватывать не имеющие к нему отношения системные исключения. В Python исключения генерируют даже такие вещи, как ошибки памяти, подлинные дефекты в программе, остановки итерации, прерывания с клавиатуры и выходы в систему. Если только вы не разрабатываете отладчик или похожий инструмент, то исключения подобного рода обычно не должны перехватываться в вашем коде.

Например, сценарии обычно завершают работу, когда управление оказывается за концом файла верхнего уровня. Однако Python также предоставляет встроенный вызов `sys.exit(`*`код-состояния`*`)`, чтобы сделать возможным ранее завершение. В действительности он работает путем генерации встроенного исключения `SystemExit` с целью окончания программы, так что обработчики `try/finally` запускаются при выходе и программы специальных видов способны перехватывать это событие [1]. По указанной причине оператор `try` с пустой конструкцией `except` может неосознанно помешать критически важному выходу, как показано в следующем файле (`exiter.py`):

```
import sys
def bye():
    sys.exit(40)              # Критическая ошибка: прекратить прямо сейчас!
try:
    bye()
except:
    print('got it')           # Ох, мы проигнорировали выход
print('continuing...')        # продолжение...
```

```
% python exiter.py
got it
continuing...
```

Вы просто можете не ожидать всех видов исключений, которые могли бы возникнуть в течение выполнения операции. В этом конкретном случае может помочь применение встроенных классов исключений из предыдущей главы, потому что суперкласс `Exception` не является суперклассом класса `SystemExit`:

```
try:
    bye()
except Exception:          # Не перехватывает выходы, но _будет_
                           #   перехватывать многие другие исключения
    ...
```

Тем не менее, в других случаях такая схема не лучше пустой конструкции `except` — из-за того, что `Exception` представляет собой суперкласс выше всех встроенных исключений кроме событий выхода в систему, он по-прежнему обладает потенциалом перехвата исключений, предназначенных для других мест в программе.

Пожалуй, хуже всего то, что использование пустой конструкции `except` и перехват суперкласса `Exception` также будут перехватывать подлинные ошибки в программе, которым должно быть разрешено проходить бóльшую часть времени. Фактически эти

---

[1] Связанный вызов `os._exit` также заканчивает программу, но через немедленное завершение — он пропускает действия очистки, в том числе любые зарегистрированные посредством упомянутого ранее модуля `atexit`, и не может быть перехвачен с помощью блоков `try/except` или `try/finally`. Обычно он используется только в порожденных дочерних процессах, рассмотрение которых выходит за рамки книги. Детали ищите в руководстве по библиотеке.

две методики способны по существу отключить механизм сообщения об ошибках интерпретатора Python, затрудняя обнаружение просчетов в коде. Взгляните на показанный далее код:

```
mydictionary = {...}
...
try:
    x = myditctionary['spam']       # Ох, опечатка
except:
    x = None                        # Предположим, мы получили KeyError
...продолжить работу с x...
```

Программист здесь предполагает, что единственным видом ошибок, которые могут произойти при индексировании словаря, будет отсутствующий ключ. Но поскольку имя myditctionary написано неправильно (должно быть mydictionary), интерпретатор Python взамен генерирует исключение NameError для ссылки на неопределенное имя, которое обработчик молча перехватывает и игнорирует. При доступе к словарю обработчик исключений будет некорректно заполнять переменную стандартным значением, маскируя ошибку в программе.

Более того, перехват Exception здесь не поможет — он будет иметь в точности тот же эффект, что и пустая конструкция except, благополучно и молча заполняя переменную стандартным значением и маскируя ошибку в программе, о которой вы наверняка хотели бы знать. Если подобное происходит в коде, который далек от того места, где применяются извлеченные значения, тогда вам предстоит очень интересная задача отладки!

В качестве эмпирического правила: будьте в своих обработчиках как можно более *конкретными* — пустые конструкции except и перехват суперкласса Exception удобны, но потенциально подвержены ошибкам. Скажем, в последнем примере было бы лучше записать except KeyError:, чтобы сделать свои намерения явными и избежать перехвата событий, не имеющих отношения к делу. В более простых сценариях вероятность возникновения проблем может оказаться не настолько значительной, чтобы перевесить удобство перехвата всех исключений, но в целом универсальные обработчики, как правило, являются источником неприятностей.

Перехват чересчур малого: используйте категории на основе классов

С другой стороны, обработчики также не должны быть слишком конкретными. Указывая специфичные исключения в try, вы перехватываете только то, что фактически перечислили. Это необязательно плохо, но если система развивается и в будущем должна перехватывать другие исключения, тогда вам может понадобиться добавлять их в списки исключений повсюду в коде.

Мы сталкивались с таким явлением в предыдущей главе. Например, приведенный ниже обработчик реализован так, чтобы трактовать MyExcept1 и MyExcept2 как нормальные случаи, а все остальное как ошибку. Однако если в будущем добавится исключение MyExcept3, то оно будет обрабатываться как ошибка, пока вы не обновите список исключений:

```
try:
    ...
except (MyExcept1, MyExcept2): # Работа нарушится, если позже добавится MyExcept3
    ...                        # Не ошибки
else:
    ...                        # Предполагается, что это ошибка
```

К счастью, аккуратное применение исключений на основе классов, которые обсуждались в главе 34, способно полностью устранить эту ловушку при сопровождении кода. Если вы перехватываете общий суперкласс, то в будущем можете добавлять и генерировать более специфичные подклассы без необходимости в ручном расширении списков исключений в конструкциях except — суперкласс становится расширяемой категорией исключений:

```
try:
    ...
except SuccessCategoryName:   # Продолжит нормально работать, если позже
                              #   добавится подкласс MyExcept3
    ...                       # Не ошибки
else:
    ...                       # Предполагается, что это ошибка
```

Другими словами, даже небольшое проектное решение проходит длинный путь. Мораль истории в том, чтобы соблюдать осторожность, не делая обработчики исключений ни слишком универсальными, ни чересчур конкретными, и разумно подбирать степень детализации операторов try. Политики в отношении исключений должны быть частью полного проектного решения, особенно в более крупных системах.

Сводка по базовому языку

Примите поздравления! На этом ознакомление с основами языка программирования Python завершено. Если вы зашли настолько далеко, то уже стали полноценным программистом на Python. В части, посвященной более сложным темам, рассматриваются дополнительные темы, которые вскоре будут описаны. Тем не менее, с точки зрения основ история о Python и главное путешествие в данной книге окончены.

В ходе чтения вы ознакомились почти со всем, что можно увидеть в самом языке, причем достаточно глубоко, чтобы применить полученные знания к большинству кода, с которым вы, вполне вероятно, столкнетесь в "диком" мире открытого кода. Вы изучили встроенные типы, операторы и выражения, а также инструменты, используемые для построения более крупных программных единиц — функций, модулей и классов. Вы исследовали важные вопросы проектирования программного обеспечения, полную парадигму ООП, инструменты функционального программирования, концепции архитектур программ, компромиссы, связанные с альтернативными инструментами, и многое другое, формируя набор навыков, который теперь можно свободно применять при разработке реальных приложений.

Комплект инструментов Python

С этого момента ваша будущая карьера, связанная с Python, будет по большому счету заключаться в овладении комплектом инструментов, доступным для программирования прикладных приложений на Python. Вы обнаружите, что это непрерывная задача. Скажем, стандартная библиотека содержит сотни модулей, а общедоступное программное обеспечение предлагает еще больше инструментов. Можно потратить десятилетия в достижении мастерства использования всех имеющихся инструментов, особенно с учетом постоянного появления новых инструментов, ориентированных на новые технологии (поверьте мне — я свыше 25 лет в данной сфере!).

Вообще говоря, Python предоставляет следующие комплекты инструментов.

Встроенные типы

Встроенные типы наподобие строк, списков и словарей позволяют быстро писать простые программы.

Расширения Python

Для более сложных задач вы можете расширять Python за счет написания собственных функций, модулей и классов.

Скомпилированные расширения

Хотя данную тему мы в книге не раскрываем, Python также можно расширять с помощью модулей, написанных на внешних языках вроде C или C++.

Поскольку Python организует свои комплекты инструментов по уровням, вы можете принимать решение о том, насколько глубоко программы должны погружаться в эту иерархию для любой задачи – применять встроенные типы для простых сценариев, добавлять расширения на Python для более крупных систем и использовать скомпилированные расширения для сложных работ. В книге были раскрыты только первые две из указанных категорий, но их достаточно, чтобы начать заниматься действительным программированием на Python.

Помимо прочих существуют инструменты, ресурсы или прецеденты применения Python практически в любой вычислительной области, какую только можно вообразить. Подсказки о том, куда двигаться дальше, были даны в обзоре приложений и пользователей Python в главе 1 первого тома. Вероятно, вы обнаружите, что с помощью такого мощного языка, как Python, решать распространенные задачи часто гораздо легче и даже приятнее, чем можно было ожидать.

Инструменты для разработки, ориентированные на более крупные проекты

Большинство примеров в книге были довольно небольшими и самодостаточными. Так было задумано, чтобы помочь вам освоить основы. Но теперь, когда вы знаете все о базовом языке, пришло время приступить к изучению способов использования встроенных и сторонних интерфейсов Python для выполнения реальной работы.

На практике программы на Python могут становиться значительно крупнее примеров, с которыми вы экспериментировали до сих пор в книге. Даже в Python наличие в нетривиальных и полезных программах *тысяч* строк кода – далеко не редкость после того, как вы соберете все индивидуальные модули в систему. Хотя основные инструменты структурирования программ на Python, подобные модулям и классам, оказывают большую помощь в управлении такой сложностью, временами дополнительную поддержку могут предложить и другие инструменты.

Для разработки крупных систем вы обнаружите такую поддержку как в Python, так и в общедоступной области. Вы видели некоторые из них в действии, и я упоминал о нескольких других. Чтобы помочь вам с дальнейшими шагами, ниже представлен краткий тур и сводка по наиболее распространенным инструментам такого рода.

PyDoc и строки документации

Функция `help` из PyDoc и HTML-интерфейсы были введены в главе 15 первого тома. PyDoc предлагает систему документации для ваших модулей и объектов, интегрируется с синтаксисом строк документации Python и является стандартной частью системы Python. Советы по источникам документации приводились в главах 15 и 4 первого тома.

PyChecker и PyLint

Поскольку Python настолько динамический язык, сообщения о некоторых ошибках в программе не появятся до тех пор, пока программа не будет запущена (даже синтаксические ошибки не улавливаются до запуска либо импортирования файла). Такую особенность нельзя считать крупным недостатком – как и с большинством языков, это всего лишь означает необходимость тестирования кода на Python перед его поставкой. В худшем случае в Python вы по существу меняете стадию компиляции на начальную стадию тестирования. Кроме того, динамическая природа Python, автоматически выдаваемые сообщения об ошибках и модель исключений позволяют найти и исправить ошибки легче и быстрее, чем в ряде других языков. Например, в отличие от С, когда встречаются ошибки, интерпретатор Python не терпит полный отказ.

Однако инструменты могут помочь и здесь. Системы PyChecker и PyLint обеспечивают поддержку для выявления часто встречающихся ошибок заблаговременно, до запуска сценария. Их роли подобны ролям программы `lint` при разработке на языке С. Некоторые разработчики на Python прогоняют свой код через PyChecker перед его тестированием или поставкой, чтобы отловить любые возможные скрытые проблемы. На самом деле первоначально имеет смысл опробовать указанные системы – ряд предупреждений, выдаваемых этими инструментами, могут помочь вам научиться выявлять и избегать распространенных недоразумений при программировании на Python. Системы PyChecker и PyLint являются сторонними пакетами с открытым кодом, доступными на веб-сайте PyPI или через предпочитаемый поисковый механизм в веб-сети. Они также могут иметь вид IDE-сред с графическими пользовательскими интерфейсами.

PyUnit (он же `unittest`)

В главе 25 первого тома было показано, как добавлять в файл Python код самотестирования за счет применения приема с `__name__ == '__main__'` в конце файла – простого протокола модульного тестирования. Для достижения более сложных целей тестирования в Python имеются два инструмента поддержки тестирования. Первый, PyUnit (называемый `unittest` в руководстве по библиотеке), предоставляет объектно-ориентированный фреймворк, основанный на классах, для указания и настройки тестовых сценариев и ожидаемых результатов. Он имитирует фреймворк JUnit для Java. Это сложно устроенная система модульного тестирования на основе классов, которая подробно описана в руководстве по библиотеке Python.

`doctest`

Стандартный библиотечный модуль `doctest` предлагает второй и более простой подход к регрессионному тестированию, базирующийся на средстве строк документации Python. Грубо говоря, для использования `doctest` вы вырезаете и вставляете содержимое журнала интерактивного сеанса тестирования в строки документации файлов исходного кода. Затем `doctest` извлекает ваши строки документации, разбивает их на тестовые сценарии и результаты и повторно прогоняет тесты для сличения их результатов с ожидаемыми. Деятельность модуля `doctest` можно настраивать разнообразными способами; полные сведения о модуле `doctest` ищите в руководстве по библиотеке.

IDE-среды

Мы обсуждали IDE-среды для Python в главе 3 первого тома. Такие IDE-среды, как IDLE, предлагают графическую среду для редактирования, выполнения, отладки и просмотра программ на Python. Ряд продвинутых IDE-сред наподобие Eclipse, Komodo, NetBeans и других перечисленных в главе 3 первого тома могут поддерживать дополнительные задачи, включая интеграцию с системой контроля версий исходного кода, рефакторинг кода, инструменты управления проектами и т.д. За более полной информацией обращайтесь в главу 3 первого тома.

Профилировщики

Из-за того, что Python является настолько высокоуровневым и динамическим, ожидания относительно производительности, основанные на опыте работе с другими языками, обычно неприменимы к коду на Python. Чтобы по-настоящему выделить в своем коде узкие места в плане производительности, понадобится добавить логику измерения времени с помощью инструментов из модуля `time` или `timeit` либо запустить код под управлением модуля `profile`. Примеры использования модулей для измерения времени при сравнении скорости выполнения итерационных инструментов и версий Python приводились в главе 21 первого тома.

Профилирование обычно будет первым шагом при оптимизации — кода для ясности, далее профиля для выявления узких мест и времени выполнения альтернативных версий для медленных частей программы. Для второго шага `profile` является стандартным библиотечным модулем, который реализует профилировщик исходного кода на Python. Он запускает предоставленную строку кода (например, импортирование файла сценария или вызов функции), после чего по умолчанию выводит в стандартный выходной поток отчет, который содержит статистические данные о производительности — количество обращений к каждой функции, время выполнения каждой функции и другую информацию.

Модуль `profile` можно запускать в виде сценария либо импортировать, а также настраивать различными путями; скажем, он способен сохранять статистику выполнения в файле, чтобы проанализировать ее позже с помощью модуля `pstats`. Для профилирования в интерактивном режиме импортируйте модуль `profile` и вызовите `profile.run('`*`код`*`')`, передав подлежащий профилированию код в форме строки (например, вызов функции, импортирование файла или код, читаемый из файла). Для профилирования в командной строке системной оболочки используйте команду вида `python -m profile main.py` *`аргументы`* (формат более подробно объясняется в приложении А). В руководствах по стандартной библиотеке Python описаны другие варианты профилирования; скажем, модуль `cProfile` имеет интерфейсы, идентичные `profile`, но сопряжен с меньшими накладными расходами, поэтому может лучше подойти для профилирования долго выполняющихся программ.

Отладчики

Варианты отладки обсуждались также в главе 3 первого тома (см. врезку "Отладка кода Python"). В качестве обзора отметим, что большинство IDE-сред для разработки на Python поддерживают отладку, основанную на графическом пользовательском интерфейсе, а стандартная библиотека Python дополнительно включает модуль для отладки исходного кода под названием pdb. Модуль pdb

предоставляет интерфейс командной строки и работает во многом подобно распространенным отладчикам для языка C (например, dbx, gdb).

Как и в случае с профилировщиком, отладчик pdb можно запускать либо интерактивно, либо из командной строки, а также импортировать и обращаться к нему в программе на Python. Для его применения в интерактивном режиме импортируйте модуль, начните выполнение кода, вызвав функцию pdb (скажем, pdb.run('main()')), и вводите команды отладки в интерактивной подсказке pdb. Для запуска pdb в командной строке системной оболочки используйте команду вида python -m pdb main.py *аргументы*. Модуль pdb также включает полезную функцию послеаварийного анализа pdb.pm(), которая запускает отладчик после того, как встретилось исключение, возможно в сочетании с флагом -i интерпретатора Python. Дополнительные сведения о таких инструментах приведены в приложении А.

Поскольку IDE-среды вроде IDLE также предлагают интерфейсы отладки в стиле "указать и щелкнуть", в наши дни pdb не считается важным инструментом кроме ситуаций, когда графический пользовательский интерфейс недоступен или желателен больший контроль. За советами по использованию графического пользовательского интерфейса IDE-среды IDLE обращайтесь в главу 3 первого тома. По правде говоря, pdb и IDE-среды не особенно часто применяются на практике — как отмечалось в главе 3, большинство программистов либо вставляют операторы print, либо просто читают сообщения об ошибках интерпретатора Python: вероятно не самый высокотехнологичный подход, но достаточно практичный, чтобы одержать победу в мире Python!

Варианты поставки

В главе 2 первого тома были представлены распространенные инструменты для упаковки программ на Python. Системы py2exe, PyInstaller и другие перечисленные в главе способны упаковывать байт-код и виртуальную машину Python в "фиксированные двоичные" автономные исполняемые файлы, которые не требуют наличия установленной копии Python на целевой машине и скрывают код вашей программы. Кроме того, программы на Python могут поставляться в формах исходного кода (.py) или байт-кода (.pyc), а привязки импортирования поддерживают специальные методики упаковки, такие как автоматическое извлечение файлов .zip и шифрование байт-кода.

Мы также кратко коснулись стандартных библиотечных модулей distutils, которые предоставляют варианты упаковки для модулей и пакетов Python и расширений, написанных на C; детали приведены в руководствах по Python. Появившаяся сторонняя система упаковки Python "eggs" (формат "яиц") предлагает еще одну альтернативу, которая также учитывает зависимости; поищите в веб-сети дополнительную информацию.

Варианты оптимизации

На тот случай, когда скорость имеет значение, имеется несколько вариантов оптимизации программ. Система PyPy, описанная в главе 2 первого тома, предоставляет оперативный компилятор для перевода байт-кода на Python в двоичный машинный код, а Shed Skin предлагает транслятор Python в C++. Иногда вы также можете встречать файлы оптимизированного байт-кода .pyo, создаваемые и запускаемые с помощью флага командной строки -O интерпретатора Python, который обсуждался в главе 22 первого тома и в главе 34 и вводится в

действие в главе 39. Тем не менее, из-за того, что такой прием обеспечивает весьма скромный рост производительности, обычно он не используется кроме как для удаления отладочного кода.

В качестве последнего средства для повышения производительности вы можете перенести части своей программы в компилируемые языки, такие как С. Расширения С более детально рассматриваются в книге *Programming Python* (`http://www.oreilly.com/catalog/9780596158101`) и в стандартных руководствах по Python. В целом показатели скорости самого интерпретатора Python также с течением времени улучшаются, поэтому обновление до более поздних выпусков может повысить производительность.

Другие советы касательно более крупных проектов

В книге встречались разнообразные базовые языковые средства, которые также станут более полезными, как только вы приступите к реализации крупных проектов. К ним относятся пакеты модулей (глава 24 первого тома), исключения на основе классов (глава 34), псевдозакрытые атрибуты классов (глава 31), строки документации (глава 15 первого тома), файлы конфигурации путей модулей (глава 22 первого тома), сокрытие имен от `from *` с помощью списков `__all__` и имена в стиле `_X` (глава 25 первого тома), добавление кода самотестирования посредством приема `__name__ == '__main__'` (глава 25 первого тома), применение общих правил проектирования для функций и модулей (главы 17, 19 и 25 первого тома), использование паттернов объектно-ориентированного проектирования (глава 31 и другие) и т.д.

Чтобы узнать о других доступных инструментах для крупномасштабной разработки на Python, просмотрите веб-сайт PyPI и веб-сеть в целом. Применение Python на самом деле представляет собой более широкую тему, чем изучение Python, поэтому вам придется обращаться к дополнительным ресурсам.

Резюме

В данной главе часть книги, посвященная исключениям, была завершена обзором концепций проектирования, распространенных сценариев использования исключений и часто применяемых инструментов для разработки.

Кроме того, глава также завершает основной материал всей книги. К настоящему моменту вы ознакомились с тем подмножеством языка Python, которое используют большинство программистов, а возможно и больше. Фактически, если вы дочитали до этого места, то можете смело себя считать *официальным программистом на Python*. Обязательно подберите себе футболку или наклейку на ноутбук с соответствующим логотипом (и не забудьте внести знание Python в свое резюме).

Следующая и последняя часть книги является набором глав, посвященных темам, которые считаются сложными, но все равно относятся к категории базового языка. Все главы финальной части предназначены для *дополнительного чтения*, потому что не каждый программист на Python обязан углубляться в их тематику, а некоторые могут отложить изучение изложенных в них вопросов до момента, когда они понадобятся. Конечно, многие из вас могут остановиться здесь и заняться исследованием ролей Python в своих предметных областях. Откровенно говоря, на практике прикладные библиотеки оказываются более важными, чем продвинутые и в чем-то даже экзотические языковые средства.

С другой стороны, если вас заботят вещи вроде Unicode или двоичных данных, имеется потребность работать с инструментами для построения API-интерфейсов, такими как дескрипторы, декораторы и метаклассы, или просто есть желание узнать дальнейшие детали, тогда следующая часть книги поможет вам начать. Довольно крупные примеры в последней части книги также предоставят вам шанс увидеть уже изученные концепции примененными более реалистичными способами.

Так как вы добрались до конца основного материала книги, то получаете возможность слегка отдохнуть от контрольных вопросов по главе — предлагается всего лишь один вопрос. Как всегда, постарайтесь проработать упражнения для данной части, чтобы закрепить знания, приобретенные в прошедших главах; поскольку следующая часть предназначена для факультативного чтения, этот набор упражнений, завершающих часть, будет последним. Если вы хотите увидеть примеры того, как все изученное ранее объединяется вместе в реальных сценариях, взятых из распространенных приложений, тогда обязательно ознакомьтесь с "решением" упражнения 4 в приложении Г.

На тот случай, если вы закончили свое путешествие в этой книге, то все же просмотрите раздел "На бис" в конце главы 41, самой последней в книге (ради читателей, продолжающих изучение следующей части, я не буду здесь раскрывать секрет).

Проверьте свои знания: контрольные вопросы

1. (Вопрос является повторением шестого контрольного вопроса из главы 1 первого тома — видите, я же говорил, что это будет легко! :-) Почему слово "spam" обнаруживается в настолько многих примерах кода Python в книгах и веб-сети?

Проверьте свои знания: ответы

1. Потому что язык Python назван в честь британской комик-группы "Монти Пайтон" (основываясь на опросах, проводимых мною в учебных группах, это слишком хорошо оберегаемая тайна в мире Python!). Слово "spam" взято из пародии "Монти Пайтон", снятой в кафетерии, где все позиции меню, похоже, включают "Spam". Пару, пытающуюся заказать еду, заглушает хор викингов, поющих о мясных консервах марки "Spam". Нет, правда. И если б я мог вставить аудиоклип этой песни здесь, то я бы...

Проверьте свои знания: упражнения для части VII

Поскольку мы добрались до конца этой части книги, самое время предложить несколько упражнений по исключениям, чтобы дать вам возможность попрактиковаться с основами. Исключения — действительно простой инструмент; если вы сумеете выполнить упражнения, то вероятно хорошо освоили область исключений. Решения упражнений приведены в приложении Г.

1. *Оператор* `try/except`. Напишите функцию по имени `oops`, которая при вызове явно генерирует исключение `IndexError`. Затем напишите еще одну функцию, вызывающую `oops` внутри оператора `try/except` для перехвата ошибки. Что произойдет, если вы измените функцию `oops`, чтобы вместо `IndexError` она генерировала `KeyError`? Откуда берутся имена `KeyError` и `IndexError`?

(Подсказка: вспомните, что все неуточненные имена обычно поступают из одной из четырех областей видимости.)

2. *Объекты и списки исключений.* Измените написанную в упражнении 1 функцию oops, чтобы она генерировала определенное вами исключение по имени MyError. Идентифицируйте свое исключение с помощью класса (если только вы не используете версию Python 2.5 или более раннюю, то обязаны поступить так). Затем расширьте оператор try в перехватывающей функции для перехвата этого исключения и его экземпляра в дополнение к IndexError, а также вывода перехваченного экземпляра.

3. *Обработка ошибок.* Напишите функцию по имени safe(func, *pargs, **kargs), которая запускает любую функцию с любым количеством позиционных и/или ключевых аргументов за счет применения заголовка произвольных аргументов * и синтаксиса вызовов, перехватывает любое исключение, сгенерированное во время выполнения функции, и выводит исключение с использованием вызова exc_info из модуля sys. Затем примените свою функцию safe для запуска функции oops из упражнения 1 или 2. Поместите функцию safe в файл модуля по имени exctools.py и передайте ей функцию oops интерактивно. Какого вида сообщения об ошибках вы получите? Наконец, расширьте функцию safe, чтобы при возникновении ошибки она также выводила трассировку стека Python путем вызова встроенной функции print_exc из стандартного модуля traceback; детали применения ищите ранее в этой главе и в справочном руководстве по библиотеке Python. Вероятно, мы могли бы реализовать safe как *декоратор функции*, используя методики из главы 32, но чтобы полностью изучить, каким образом, придется перейти к следующей части книги (за предварительным обзором обращайтесь к решениям упражнений).

4. *Примеры для самообучения.* В конце приложения Г я привожу несколько примеров сценариев, которые разработаны как групповые упражнения в существующих классах Python для изучения вами и самостоятельного запуска в сочетании со стандартным набором руководств по Python. Они не описаны и применяют инструменты из стандартной библиотеки Python, которые вы должны исследовать самостоятельно. Тем не менее, для многих читателей это помогает увидеть, как обсуждаемые в книге концепции объединяются в реальных программах. Если они еще больше разожгут у вас интерес, то вы можете найти множество более крупных и реалистичных примеров программ на Python прикладного уровня в книгах наподобие *Programming Python* (http://www.oreilly.com/catalog/9780596158101) и в веб-сети.

ЧАСТЬ VIII

Более сложные темы

ГЛАВА 37

Unicode и байтовые строки

До сих пор наше исследование строк в книге было намеренно неполным. При предварительном обзоре типов в главе 4 первого тома были кратко представлены строки и файлы Unicode языка Python без многочисленных деталей, а в главе 7 первого тома, посвященной *строкам*, их границы были умышленно сужены до подмножества тем о строках, которые необходимо знать большинству программистов на Python.

Так было задумано: поскольку многие программисты, включая большинство новичков, имеют дело с простыми формами текста наподобие ASCII, они могут благополучно работать с базовым строковым типом `str` в Python и ассоциированными с ним операциями, и не нуждаются в том, чтобы разбираться с более сложными концепциями строк. Фактически такие программисты часто могут игнорировать изменения в строках, внесенные в Python 3.X, и продолжать использовать строки, как они поступали в прошлом.

С другой стороны, многие другие программисты занимаются более специализированными типами данных: наборами символов, отличающимися от ASCII, содержимым файлов изображений и т.д. Для этих программистов и остальных, кто может когда-нибудь к ним присоединиться, мы собираемся здесь дополнить историю со строками Python и взглянуть на ряд более сложных концепций в строковой модели Python.

В частности, мы исследуем основы поддержки Python для *текста Unicode* — обогащенных символьных строк, применяемых в интернационализированных приложениях, а также *двоичных данных* — строк, которые представляют абсолютные байтовые значения. Как мы увидим, представление расширенных строк в последних версиях Python разошлось.

- Python 3.X предоставляет для двоичных данных альтернативный строковый тип и в нормальном строковом типе поддерживает текст Unicode (включая ASCII).
- Python 2.X предоставляет для отличающегося от ASCII текста Unicode альтернативный строковый тип и в нормальном строковом типе поддерживает как простой текст, так и двоичные данные.

Вдобавок поскольку строковая модель Python напрямую влияет на то, как обрабатываются *файлы* не ASCII, мы также исследуем основы этой связанной темы. Наконец, мы кратко рассмотрим расширенные *инструменты* для работы со строками и двоичными данными, такие как сопоставление с образцом, обеспечение постоянства объ-

ектов посредством модуля pickle, упаковка двоичных данных и разбор XML, а также влияние на них изменений в строках Python 3.X.

Глава официально считается посвященной сложным темам, потому что не всем программистам понадобится погружаться в мир кодировок Unicode или двоичных данных. Одним читателям может хватить предварительного обзора из главы 4 первого тома, другие могут оставить эту главу для чтения в будущем. Однако если у вас когда-либо возникнет необходимость в обработке текста Unicode или двоичных данных, тогда вы обнаружите, что строковая модель Python предлагает нужную поддержку.

Изменения строк в Python 3.X

Одним из наиболее заметных изменений в линейке Python 3.X стала мутация типов строковых объектов. Вкратце типы str и unicode из Python 2.X были трансформированы в типы bytes и str в Python 3.X плюс добавился новый изменяемый тип bytearray. Формально тип bytearray доступен также в Python 2.6 и 2.7 (хотя не в более ранних версиях), но он был перенесен из Python 3.X и не настолько ясно проводит различие между текстовым и двоичным содержимым в Python 2.X.

Такие изменения способны оказать значительное влияние на ваш код, особенно если вы обрабатываете данные, по своей природе имеющие вид Unicode или двоичный. В качестве эмпирического правила следует отметить, что степень важности для вас этой темы во многом зависит от того, под какую из перечисленных ниже категорий вы подпадаете.

- Если вы имеете дело с текстом Unicode, отличным от ASCII — например, в контексте интернационализированных областей вроде веб-сети или результатов, получаемых от некоторых инструментов разбора и баз данных XML и JSON, — тогда вы обнаружите, что поддержка кодировок текста в Python 3.X отличается, но также является, вероятно, более прямой, понятной и бесшовной по сравнению с Python 2.X.
- Если вы имеете дело с двоичными данными — скажем, в форме файлов с изображениями или аудиоклипами, либо упакованных данных, обрабатываемых с помощью модуля struct, — то вам нужно будет освоить новый объект bytes, введенный в Python 3.X, а также осмыслить другое и более резкое отличие между текстовыми и двоичными данными и файлами в Python 3.X.
- Если вы не относитесь ни к одной из предшествующих двух категорий, тогда обычно можете работать со строками в Python 3.X почти как поступали бы в Python 2.X, используя универсальный строковый тип str, текстовые файлы и все привычные строковые операции, которые изучались ранее. Интерпретатор Python 3.X будет кодировать и декодировать строки с применением стандартной кодировки вашей платформы (например, ASCII, UTF-8, or Latin-1 — узнать ее можно через вызов locale.getpreferredencoding(False)), но вряд ли вы это заметите.

Другими словами, если вы всегда работаете с текстом ASCII, то можете обойтись нормальными строковыми объектами и текстовыми файлами, пока избегая большей части следующей истории. Как вскоре будет показано, ASCII является простой разновидностью Unicode и подмножеством других кодировок, так что строковые операции и файлы в целом "просто работают", если ваши программы обрабатывают только текст ASCII.

Тем не менее, даже если вы подпадаете под третью из упомянутых выше категорий, то понимание основ Unicode и строковой модели Python 3.X поможет прояснить лежащее в основе поведение теперь и упростить решение проблем с Unicode или двоичными данными, если они окажут воздействие в будущем.

Выражаясь более строго: нравится это или нет, но Unicode будет частью разработки большинства программного обеспечения в будущем, основы которого мы заложили, и со временем наверняка коснется и вас. Несмотря на то что приложения выходят за рамки материала книги, если вы работаете с веб-сетью, файлами, каталогами, сетевыми интерфейсами, базами данных, каналами, JSON, XML и даже графическими пользовательскими интерфейсами, то Unicode перестает быть необязательной темой в Python 3.X.

Поддержка Python 3.X для Unicode и двоичных данных также доступна в Python 2.X, хотя и в других формах. Невзирая на то что основное внимание в главе сосредоточено на строковых типах в Python 3.X, для читателей, использующих Python 2.X, мы попутно исследуем, чем будет отличаться эквивалентная поддержка в Python 2.X. Безотносительно к применяемой версии Python, описываемые здесь инструменты могут стать важными во многих типах программ.

Основы строк

Прежде чем мы обратимся к коду, давайте начнем с общего обзора строковой модели Python. Чтобы понять, почему в Python 3.X изменился подход в этом направлении, мы должны выяснить, как на самом деле символы представляются в компьютерах — когда они закодированы в файлах и когда хранятся в памяти.

Схемы кодирования символов

Большинство программистов думают о строках как о последовательностях символов, используемых для представления текстовых данных. Несмотря на точность такой формулировки, способ хранения символов может варьироваться в зависимости от того, какой набор символов должен быть записан. Например, когда текст хранится в файлах, применяемый в нем набор символов определяет его формат.

Наборы символов являются стандартами, которые назначают индивидуальным символам целочисленные коды, чтобы они могли быть представлены в памяти компьютера. Скажем, стандарт *ASCII* был создан в США и формирует понятие текстовых строк для многих программистов из США. Стандарт ASCII определяет коды символов от 0 до 127 и делает возможным хранение каждого символа в одном 8-битном байте, в котором фактически используются только 7 битов.

Например, стандарт ASCII отображает символ 'a' на целочисленное значение 97 (шестнадцатеричное 0x61), которое может храниться внутри единственного байта в памяти и файлах. Если вы хотите посмотреть, как это работает, то встроенная функция ord в Python дает двоичное идентифицирующее значение для символа, а chr возвращает символ для заданного значения целочисленного кода:

```
>>> ord('a')  # 'a' - байт, закодированный как значение 97
              #  в стандарте ASCII (и других)
97
>>> hex(97)
'0x61'
>>> chr(97)   # В ASCII значение целочисленного кода 97 обозначает символ 'a'
'a'
```

Однако иногда одного байта на символ недостаточно. Скажем, разнообразные символы и диакритические знаки не умещаются в диапазон возможных символов, определяемых ASCII. Чтобы приспособиться к специальным символам, некоторые стандарты для представления символов применяют все допустимые значения в 8-битном байте, 0–255, и назначают специальным символам значения от 128 до 255 (за пределами диапазона ASCII).

Один такой стандарт, известный как набор символов *Latin-1*, широко используется в Западной Европе. В стандарте Latin-1 коды символов выше 127 назначаются диакритическим знакам и специальным символам. Например, байтовое значение 196 соответствует специально помеченному символу, не входящему в набор ASCII:

```
>>> 0xC4
196
>>> chr(196)      # Показана результирующая форма в Python 3.X
'Ä'
```

Стандарт Latin-1 делает возможным представление обширного комплекта дополнительных специальных символов, но по-прежнему поддерживает ASCII как 7-битное подмножество своей 8-битной реализации.

Тем не менее, в ряде алфавитов определено настолько много символов, что представить каждый из них как один байт попросту невозможно. *Unicode* обеспечивает более высокую гибкость. Временами на текст Unicode ссылаются как на строки "широких символов", потому что при необходимости символы могут быть представлены посредством множества байтов. Unicode обычно применяется в *интернационализированных* программах для представления европейских, азиатских и других неанглийских наборов символов, которые имеют больше символов, чем способны представить 8-битные байты.

Чтобы хранить обогащенный текст подобного рода в памяти компьютера, такие символы нужно транслировать в низкоуровневые байты с использованием *кодировки*. Кодировка определяет правила для перевода строки символов Unicode в последовательность байтов и извлечения строки из последовательности байтов. Более формально трансляция туда и обратно между байтами и строками определяется двумя терминами:

- кодирование – это процесс перевода строки символов в форму низкоуровневых байтов согласно заданному имени кодировки;
- декодирование – это процесс перевода низкоуровневой строки байтов в форму строки символов согласно ее имени кодировки.

То есть мы *кодируем* строку в низкоуровневые байты и *декодируем* низкоуровневые байты в строку. Для сценариев декодированные строки являются просто символами в памяти, но могут быть закодированы в разнообразные представления байтовых строк при хранении в файлах, передаче по сетям, внедрении в документы и базы данных и т.д.

В некоторых кодировках процесс трансляции тривиален – скажем, ASCII и Latin-1 отображают каждый символ на одиночный байт *фиксированного размера*, так что никакой работы по переводу не требуется. Для других кодировок отображение может быть более сложным и выдавать множество байтов на символ даже в случае простых 8-битных форм текста.

Например, широко применяемая кодировка *UTF-8* позволяет представлять большой диапазон символов за счет использования схемы с *переменным* количеством байтов. Символы с кодами ниже 128 представляются как один байт; символы с кодами

между 128 и 0x7ff (2047) переводятся в 2 байта, где каждый байт имеет значение 128–255, а символы с кодами выше 0x7ff переводятся в 3- или 4-байтовые последовательности со значениями байтов 128–255. Это позволяет сохранять простые строки ASCII компактными, обходить вопросы упорядочения байтов и избегать нулевых байтов, которые могут вызывать проблемы при работе с библиотеками C и сетями.

Поскольку кодировки Latin-1 и UTF-8 ради совместимости отображают назначенные символы на те же самые коды, ASCII является *подмножеством* указанных кодировок. То есть допустимая строка символов ASCII также будет допустимой строкой, закодированной с помощью Latin-1 и UTF-8. Скажем, каждый файл ASCII считается допустимым файлом UTF-8, т.к. набор символов ASCII представляет собой 7-битное подмножество UTF-8.

И наоборот, кодировка UTF-8 совместима в двоичном виде с ASCII, но только для кодов символов меньше 128. Кодировки Latin-1 и UTF-8 просто дают возможность представлять дополнительные символы: Latin-1 – символы, отображаемые на значения от 128 до 255 внутри байта, а UTF-8 – символы, которые могут быть представлены посредством множества байтов.

Другие кодировки позволяют представлять более широкие наборы символов различными способами. Например, *UTF-16* и *UTF-32* форматируют текст со схемой соответственно 2 и 4 байта фиксированного размера на каждый символ даже для символов, которые иначе уместились бы в один байт. Некоторые кодировки могут также вставлять префиксы, идентифицирующие порядок следования байтов.

Чтобы увидеть все самостоятельно, вызовите метод encode строки, который выдает ее в формате байтовой строки, закодированной в соответствии с указанной схемой – 2-символьная строка ASCII занимает 2 байта в ASCII, Latin-1 и UTF-8, но она намного шире в UTF-16 и UTF-32, к тому же включает заголовочные байты:

```
>>> S = 'ni'
>>> S.encode('ascii'), S.encode('latin1'), S.encode('utf8')
(b'ni', b'ni', b'ni')

>>> S.encode('utf16'), len(S.encode('utf16'))
(b'\xff\xfen\x00i\x00', 6)

>>> S.encode('utf32'), len(S.encode('utf32'))
(b'\xff\xfe\x00\x00n\x00\x00\x00i\x00\x00\x00', 12)
```

В Python 2.X результаты слегка отличаются (вы не получите ведущий символ b для байтовых строк). Но все эти схемы кодирования – ASCII, Latin-1, UTF-8 и многие другие – считаются Unicode.

При программировании на Python кодировки указываются как строки, содержащие имя кодировки. Python поступает с приблизительно сотней разных кодировок; полный список ищите в справочном руководстве по библиотеке Python. Импортирование модуля encodings и выполнение help(encodings) также покажет многие имена кодировок; одни реализованы в Python, другие в C. Кроме того, некоторые кодировки имеют несколько имен; например, *latin-1*, *iso_8859_1* и *8859* – синонимы кодировки Latin-1. Мы еще вернемся к кодировкам позже в главе, когда будем исследовать методики написания строк Unicode в сценарии.

За дополнительными сведениями о кодировке Unicode обращайтесь в стандартный набор руководств по Python. Он содержит подраздел "Unicode HOWTO" внутри раздела "Python HOWTOs", где вы найдете информацию, которая ради экономии места здесь было опущена.

Хранение строк Python в памяти

Кодировки из предыдущего раздела на самом деле применяются, только когда текст хранится или передается внешне, в файлах или на других носителях. В памяти Python всегда хранит декодированные строки текста в *нейтральном к кодировкам* формате, который может использовать или не использовать множество байтов для каждого символа. Вся обработка текста происходит в таком унифицированном внутреннем формате. Текст транслируется в специфичный для кодировки формат и обратно только при передаче во внешние текстовые файлы, байтовые строки либо API-интерфейсы с особыми требованиями к кодировке либо из них. Они являются просто строковыми объектами, представленными в настоящей книге.

Хотя приведенная далее информация не имеет отношения к коду, она может помочь некоторым читателям лучше осознать происходящее. Способ действительного хранения текста в памяти подвержен изменениям с течением времени и фактически он претерпел значительную мутацию, начиная с версии Python 3.3.

Python 3.2 и более ранние версии

Вплоть до версии Python 3.2 строки хранились внутренне в формате с *фиксированной длиной* UTF-16 (грубо говоря, UCS-2) с 2 байтами на символ, если только Python не был сконфигурирован на выделение 4 байтов на символ (UCS-4).

Python 3.3 и более поздние версии

В Python 3.3 и последующих версиях взамен применяется схема с *переменной длиной* с 1, 2 или 4 байтами на символ в зависимости от содержимого строки. Размер выбирается на основе символа с наибольшим порядковым значением Unicode в представленной строке. Такая схема делает возможным эффективное в отношении пространства представление для распространенных случаев, но также позволяет использовать полный формат UCS-4 на всех платформах.

Новая схема, появившаяся в Python 3.3, является оптимизацией, особенно в сравнении с прошлыми широкими формами Unicode. Согласно документации по Python, объем занимаемой памяти делится на 2-4 в зависимости от текста; кодирование строки ASCII в UTF-8 больше не нуждается в перекодировании символов, потому что ее представления ASCII и UTF-8 одинаковы; повторение одиночной буквы ASCII и получение подстроки из строки ASCII выполняется в 4 раза быстрее; UTF-8 оказывается в 2-4 раза быстрее; и кодирование UTF-16 — до 10 раз быстрее. Некоторые эталонные тесты показывают, что общее потребление памяти в Python 3.3 стало в 2-3 раза ниже, чем в Python 3.2, и похоже на менее ориентированную на Unicode версию Python 2.7.

Независимо от используемой схемы хранения, как отмечалось в главе 6 первого тома, Unicode очевидно требует от нас думать о строках в терминах *символов*, а не *байтов*. Это может стать более крупным препятствием для программистов, привыкших к более простому миру с одной лишь кодировкой ASCII, где каждый символ отображается на одиночный байт, но такая идея больше неприменима с точки зрения как результатов вызова инструментов для работы с текстовыми строками, так и физического размера символов.

Инструменты обработки текста

В наши дни содержимое и длина строки в действительности выражаются в *кодовых точках* Unicode — идентифицирующих порядковых числах для символов. Например, встроенная функция `ord` теперь возвращает порядковое значение

кодовой точки Unicode символа, которое не обязательно будет кодом ASCII и может умещаться или не умещаться в один 8-битный байт. Аналогично `len` возвращает количество символов, не байтов; строка, вероятно, занимает больше места в памяти, а ее символы могут не умещаться в байты.

Размер текста

Как демонстрировалось в примерах главы 4 первого тома, одиночный символ в Unicode не обязательно отображается напрямую на одиночный байт, либо при кодировании в файле, либо при хранении в памяти. Даже символы простого текста в 7-битном ASCII могут не отображаться на байты — UTF-16 использует множество байтов на символ в файлах, а Python может выделять 1, 2 или 4 байта на символ в памяти. Мышление в терминах символов позволяет нам абстрагироваться от деталей внешнего и внутреннего хранения.

Однако основной момент здесь в том, что *кодирование* имеет отношение главным образом к файлам и передаче. После загрузки в строку Python с текстом в памяти не связано понятие "кодировка" и он является просто последовательностью символов Unicode (известных как кодовые точки), хранящихся обобщенным образом. В сценарии обращение к такой строке осуществляется как к строковому объекту Python — тема следующего раздела.

Типы строк Python

На более конкретном уровне язык Python предлагает строковые типы данных для представления символьного текста в сценариях. Применяемые в сценариях строковые типы зависят от используемой версии Python. В *Python 2.X* имеется универсальный строковый тип для представления двоичных данных и простого 8-битного текста вроде ASCII, а также специфический тип для представления обогащенного текста Unicode:

- `str` для представления 8-битного текста и двоичных данных;
- `unicode` для представления декодированного текста Unicode.

Два строковых типа Python 2.X отличаются (`unicode` разрешает некоторым символам Unicode иметь добавочный размер, также обладает дополнительной поддержкой для кодирования и декодирования), но их наборы операций значительно перекрываются. Строковый тип `str` в Python 2.X применяется для текста, который может быть представлен с помощью 8-битных байтов (включая ASCII и Latin-1), равно как двоичных данных, представляющих абсолютные байтовые значения.

По контрасту с этим *Python 3.X* поступает с тремя типами строковых объектов — один для текстовых данных и два для двоичных данных:

- `str` для представления декодированного текста Unicode (в том числе ASCII);
- `bytes` для представления двоичных данных (включая декодированный текст);
- `bytearray`, изменяемая разновидность типа `bytes`.

Как упоминалось ранее, тип `bytearray` также доступен в Python 2.6 и 2.7, но он просто перенесен из Python 3.X, имеет менее специфичное к содержимому поведение и в целом считается типом Python 3.X.

Для чего нужны разные строковые типы?

Все три строковых типа в Python 3.X поддерживают похожие наборы операций, но они исполняют разные роли. Главной целью такого изменения в Python 3.X было *объединение* нормальных строковых типов и строковых типов Unicode, определяемых Python 2.X, в единственный строковый тип, который поддерживает простой текст и текст Unicode: разработчики хотели устранить противопоставление между строками Python 2.X и сделать обработку Unicode более естественной. С учетом того, что ASCII и другой 8-битный текст на самом деле являются простым видом Unicode, подобное сближение выглядело вполне логичным.

Для достижения этого в Python 3.X текст хранится в заново определенном типе `str` — *неизменяемой последовательности символов* (не обязательно байтов). Последовательность может содержать либо простой текст, такой как ASCII со значениями символов, умещающимися в одиночный байт, либо обогащенный текст наподобие UTF-8 со значениями символов, которые могут требовать нескольких байтов. Строки, обрабатываемые сценарием посредством типа `str`, хранятся в памяти обобщенным образом и кодируются в байтовые строки и декодируются из них согласно либо стандартной для платформы кодировке Unicode, либо явно указанному имени кодировки. В итоге сценарии получают возможность транслировать текст в различные кодировки, как для сохранения в памяти, так и при взаимодействии с файлами.

Хотя новый тип `str` в Python 3.X добивается желательного объединения строк/`unicode`, многим программам по-прежнему необходимо обрабатывать низкоуровневые двоичные данные, которые не закодированы в каком-либо текстовом формате. К такой категории относятся файлы с изображениями и аудиоклипами плюс упакованные данные, используемые для взаимодействия с устройствами или программами С, которые вы можете обрабатывать с помощью модуля `struct` библиотеки Python. Поскольку строки Unicode декодируются *из* байтов, они не могут применяться для представления байтов.

Для поддержки обработки таких по-настоящему двоичных данных также был введен новый строковый тип `bytes` — *неизменяемая последовательность 8-битных целых чисел*, представляющих абсолютные байтовые значения, которые по возможности выводятся как символы ASCII. Несмотря на то что `bytes` является отдельным типом, он поддерживает почти все те же операции, что и тип `str`; сюда входят строковые методы, операции над последовательностями и даже сопоставление с образцом из модуля `re`, но не строковое форматирование. В Python 2.X эту роль для двоичных данных удовлетворяет универсальный тип `str`, т.к. его строки являются всего лишь последовательностями байтов; отдельные тип `unicode` обрабатывает обогащенные текстовые строки.

Обратимся к деталям: объект `bytes` в Python 3.X на самом деле представляет собой последовательность коротких целых чисел, каждое из которых находится в диапазоне 0–255; индексирование `bytes` возвращает `int`, нарезание `bytes` возвращает еще один объект `bytes`, а выполнение встроенной функции `list` на `bytes` возвращает список целых чисел, не символов. Тем не менее, при обработке посредством операций, которые рассчитывают на символы, содержимое объектов `bytes` трактуется как байты, закодированные посредством ASCII (например, метод `isalpha` предполагает, что каждый байт является кодом символа ASCII). Более того, для удобства объекты `bytes` выводятся как строки символов, а не целые числа.

Работая над изменением строковых типов в Python 3.X, разработчики также добавили тип `bytearray`, который является *изменяемым* вариантом типа `bytes` и потому поддерживающим изменения на месте. Подобно типам `str` и `bytes` он поддерживает

обычные строковые операции и многие операции изменения на месте, характерные для списков (например, методы append и extend, а также присваивание по индексу). Тип bytearray может быть удобен при работе с действительными двоичными данными и простыми видами текста. Исходя из предположения, что ваши текстовые строки могут трактоваться как низкоуровневые 8-битные байты (скажем, текст ASCII или Latin-1), тип bytearray в заключение добавляет прямую изменяемость на месте для текстовых данных, которая временами невозможна без преобразования в изменяемый тип в Python 2.X и поддерживается типом str или bytes в Python 3.X.

Несмотря на то что Python 2.X и Python 3.X предлагают почти ту же самую функциональность, они упаковывают ее по-разному. Строковые типы Python 2.X отображаются на строковые типы Python 3.X не полностью прямо. Дело в том, что тип str из Python 2.X приравнивается к типам str и bytes в Python 3.X, а тип str из Python 3.X приравнивается к типам str и unicode в Python 2.X. Кроме того, изменяемость типа bytearray в Python 3.X уникальна.

Однако по существу эта асимметрия не так страшна, как может показаться. Она сводится к следующему: в Python 2.X вы будете использовать тип str для простых текстовых и двоичных данных и тип unicode для расширенных форм текста, чьи наборы символов не отображаются на 8-битные байты; в Python 3.X вы будете применять str для любой разновидности текста (ASCII, Latin-1 и все остальные виды Unicode) и bytes или bytearray для двоичных данных. На практике выбор часто делается за вас используемыми инструментами — особенно в случае инструментов обработки файлов, которые рассматриваются ниже.

Текстовые и двоичные файлы

Файловый ввод-вывод в Python 3.X также был модернизирован, чтобы отражать разграничение str/bytes и автоматически поддерживать кодирование текста Unicode при передачах. Теперь в Python 3.X проведено четкое и независимое от платформы различие между текстовыми и двоичными файлами.

Текстовые файлы

Когда файл открывается в *текстовом режиме*, чтение его данных автоматически декодирует его содержимое и возвращает его как объект str; запись берет объект str и автоматически кодирует его перед передачей в файл. Операции чтения и записи выполняют трансляцию в соответствии со стандартной кодировкой для платформы или указанным именем кодировки. В текстовом режиме файлы также поддерживают универсальный перевод признака конца файла и дополнительные аргументы спецификации кодировки. В зависимости от имени кодировки текстовые файлы могут также автоматически обрабатывать последовательность в начале файла, обозначающую порядок следования байтов (рассматривается далее в главе).

Двоичные файлы

Когда файл открывается в *двоичном режиме* за счет добавления b (только в нижнем регистре) к аргументу строки режима во встроенном вызове open, чтение его данных их не декодирует, а возвращает в низкоуровневом и неизмененном виде как объект bytes; подобным же образом запись берет объект bytes и передает его в файл без изменений. В двоичном режиме файлы также принимают объект bytearray для содержимого, подлежащего записи в файл.

Поскольку язык проводит четко разграничение между `str` и `bytes`, вы должны решить, являются ли ваши данные текстовыми или двоичными по своей природе, и применять объекты типа либо `str`, либо `bytes` для надлежащего представления содержимого в сценарии. В конечном итоге режим, в котором вы открываете файл, будет диктовать тип объекта, используемый в сценарии для представления его содержимого.

- Если вы обрабатываете файлы изображений, данные, передаваемые по сети, упакованные двоичные данные, содержимое которых должно быть извлечено, или потоки данных из устройств, тогда велики шансы, что вы захотите применять тип `bytes` и файлы в двоичном режиме. Вы также можете выбрать тип `bytearray`, если пожелаете обновлять данные, не создавая их копии в памяти.
- Если взамен вы обрабатываете что-то текстовое по своей природе, такое как вывод программы, HTML-разметка, содержимое сообщения электронной почты либо файлы CSV или XML, тогда вероятно захотите использовать тип `str` и файлы в текстовом режиме.

Обратите внимание, что аргумент *строки режима* для встроенной функции `open` (второй аргумент) в Python 3.X становится довольно важным — его содержимое не только задает *режим обработки* файла, но также подразумевает *объектный тип* Python. Добавляя `b` в строку режима, вы указываете двоичный режим и будете получать или обязаны предоставлять объект `bytes` для представления содержимого файла при чтении или записи. В случае отсутствия `b` файл обрабатывается в текстовом режиме, и для представления его содержимого в сценарии вы будете применять объекты `str`. Например, режимы `rb`, `wb` и `rb+` подразумевают тип `bytes`, а `r`, `w+` и `rt` (по умолчанию) — тип `str`.

Файлы в текстовом режиме также обрабатывают последовательность с *маркером порядка следования байтов* (byte order mark — BOM), которая может появляться в начале файлов в условиях ряда схем кодирования. Скажем, в кодировках UTF-16 и UTF-32 маркер BOM указывает формат с обратным или прямым порядком байтов (по существу, какой из концов битовой строки более значащий) — в качестве примеров просмотрите ведущие байты в результатах вызовов кодирования UTF-16 и UTF-32 ранее в главе. Текстовый файл UTF-8 также может включать маркер BOM для объявления о том, что в целом он относится к UTF-8. При чтении и записи данных с использованием этих схем кодирования интерпретатор Python пропускает или записывает маркер BOM в соответствии с правилами, которые приводятся позже в главе.

В Python 2.X поддерживает то же самое поведение, но для доступа к данным, основанным на байтах, применяются нормальные файлы, которые открываются посредством `open`, а для обработки текстовых данных Unicode используются файлы Unicode, открываемые с помощью вызова `codecs.open`. Как мы увидим далее в главе, во втором случае при передаче также выполняется кодирование и декодирование. Для начала мы займемся исследованием строковой модели Unicode в Python.

Написание базовых строк

Давайте рассмотрим несколько примеров, которые продемонстрируют применение строковых типов Python 3.X. Одно предварительное замечание: код в текущем разделе выполняется только под управлением Python 3.X. Тем не менее, базовые строковые операции большей частью переносимы между версиями Python. Простые стро-

ки ASCII, представляемые посредством типа `str`, в Python 2.X и Python 3.X работают одинаково (и в точности, как было описано в главе 7 первого тома).

Более того, хотя тип `bytes` в Python 2.X отсутствует (есть лишь универсальный тип `str`), под управлением Python 2.X обычно можно запускать код, где предполагается его наличие. В версиях Python 2.6 и 2.7 вызов `bytes(X)` представляет собой синоним `str(X)`, а новая литеральная форма `b'...'` считается той же самой, что и нормальный строковый литерал `'...'`. Однако в отдельных случаях вы все еще можете столкнуться с нестыковкой версий; например, вызов `bytes` в Python 2.6/2.7 не требует и не разрешает передавать второй аргумент (имя кодировки), который обязателен в `bytes` из Python 3.X.

Строковые литералы Python 3.X

Строковые объекты Python 3.X порождаются, когда вы вызываете встроенную функцию, такую как `str` или `bytes`, читаете файл, созданный вызовом `open` (описан в следующем разделе), либо используете литеральный синтаксис в своем сценарии. В Python 3.X для создания объектов `bytes` применяется новая литеральная форма `b'xxx'` (и эквивалентная ей `B'xxx'`), а объекты `bytearray` можно создавать вызовом функции `bytearray` с различными аргументами.

Выражаясь более формально, в Python 3.X все текущие формы строковых литералов — `'xxx'`, `"xxx"` и блоки в утроенных кавычках — генерируют объект `str`; добавление `b` или `B` перед любой из них приводит к созданию объекта `bytes`. Новый байтовый литерал `b'...'` по своей форме похож на низкоуровневую строку `r'...'`, используемую для того, чтобы избежать отмены специального значения символов обратной косой черты. Взгляните на следующее взаимодействие в Python 3.X:

```
C:\code> C:\python37\python
>>> B = b'spam'             # Байтовый литерал Python 3.X создает объект bytes
                            #   (8-битовые байты)
>>> S = 'eggs'              # Строковый литерал Python 3.X создает объект
                            #   текстовой строки Unicode

>>> type(B), type(S)
(<class 'bytes'>, <class 'str'>)

>>> B                       # bytes: последовательность целых чисел,
                            #   выводится как строка символов
b'spam'
>>> S
'eggs'
```

Объект `bytes` в Python 3.X в действительности является последовательностью коротких целых чисел, хотя он выводит свое содержимое в виде символов всякий раз, когда это возможно:

```
>>> B[0], S[0]              # Индексирование возвращает целое число
                            #   для bytes и строку для str
(115, 'e')
>>> B[1:], S[1:]            # Нарезание создает еще один объект bytes или str
(b'pam', 'ggs')
>>> list(B), list(S)
([115, 112, 97, 109], ['e', 'g', 'g', 's'])  # На самом деле bytes -
                                             # 8-битовые короткие целые числа
```

Объект `bytes` также неизменяем почти как `str` (но описываемый позже `bytearray` – нет); присваивать по смещению `bytes` объект `str`, `bytes` или целого числа нельзя:

```
>>> B[0] = 'x'            # Оба неизменяемы
TypeError: 'bytes' object does not support item assignment
Ошибка типа: объект bytes не поддерживает присваивание в отношении элементов
>>> S[0] = 'x'
TypeError: 'str' object does not support item assignment
Ошибка типа: объект str не поддерживает присваивание в отношении элементов
```

Наконец, обратите внимание на то, что префикс `b` или `B` литерала `bytes` также работает с любой формой строкового литерала, включая блоки в утроенных кавычках, несмотря на то, что вы получаете строку низкоуровневых байтов, которые могут отображаться или не отображаться на символы:

```
>>> # Префикс литерала bytes работает с одинарными, двойными,
>>> # утроенными кавычками и низкоуровневыми байтами
>>> B = B"""
... xxxx
... yyyy
... """
>>> B
b'\nxxxx\nyyyy\n'
```

Литералы Unicode из Python 2.X, начиная с версии Python 3.3

Формы строковых литералов Unicode из Python 2.X вида `u'xxx'` и `U'xxx'` были удалены в версии Python 3.0, потому что их сочли избыточными – нормальные строки Python 3.X представлены в кодировке Unicode. Тем не менее, для содействия прямой и обратной совместимости они снова стали доступными в версии Python 3.3, где трактуются как нормальные строки `str`:

```
C:\code> C:\python37\python
>>> U = u'spam'  # Литерал Unicode из Python 2.X воспринимается в Python 3.3+
>>> type(U)      # Он является просто строкой str, но обратно совместим
<class 'str'>
>>> U
'spam'
>>> U[0]
's'
>>> list(U)
['s', 'p', 'a', 'm']
```

Эти литералы отсутствовали в версиях Python 3.0–3.2, где взамен нужно было применять `'xxx'`. В целом вы должны использовать текстовые литералы `'xxx'` из Python 3.X в новом коде, ориентированном только на Python 3.X, поскольку форма из Python 2.X избыточна. Однако в Python 3.3 и последующих версиях применение литеральной формы из Python 2.X может облегчить задачу переноса кода Python 2.X и повысить совместимость кода Python 2.X (просмотрите пример с форматированием денежных значений, который был приведен в главе 25 первого тома и упоминается ниже). Тем не менее, безотносительно к тому, как текстовые строки записываются в Python 3.X, все они представлены в Unicode, даже если содержат только символы ASCII (см. раздел "Написание текста, отличающегося от ASCII" далее в главе).

Строковые литералы Python 2.X

Все три формы строковых литералов Python 3.X, приведенные в предыдущем разделе, могут быть записаны в Python 2.X, но их смысл отличается. Ранее упоминалось, что в версиях Python 2.6 и 2.7 байтовый литерал b'xxx' присутствует для прямой совместимости с Python 3.X, но он такой же, как 'xxx', и создает объект str (b игнорируется), а bytes представляет собой всего лишь синоним для str. Вы уже видели, что в Python 3.X оба они относятся к отдельному типу bytes:

```
C:\code> C:\python27\python
>>> B = b'spam'   # Байтовый литерал Python 3.X - это просто str в Python 2.6/2.7
>>> S = 'eggs'    # str является последовательностью байтов/символов

>>> type(B), type(S)
(<type 'str'>, <type 'str'>)
>>> B, S
('spam', 'eggs')
>>> B[0], S[0]
('s', 'e')
>>> list(B), list(S)
(['s', 'p', 'a', 'm'], ['e', 'g', 'g', 's'])
```

В Python 2.X специальный литерал и тип Unicode умещают в себе обогащенные формы текста:

```
>>> U = u'spam'    # Литерал Unicode из Python 2.X создает отдельный тип
>>> type(U)   # Работает также, начиная с Python 3.3, но является там просто str
<type 'unicode'>
>>> U
u'spam'
>>> U[0]
u's'
>>> list(U)
[u's', u'p', u'a', u'm']
```

Как мы видели, для совместимости такая форма работает также в Python 3.3 и последующих версиях, но просто создает там нормальный объект str (u игнорируется).

Преобразования строковых типов

Несмотря на то что в Python 2.X объекты типов str и unicode разрешено смешивать в выражениях (когда объект str содержит только 7-битный текст ASCII), в Python 3.X между ними проводится более четкое различие — объекты типов str и bytes *никогда* автоматически не смешиваются в выражениях и *никогда* автоматически не преобразуются друг в друга при передаче в функции. Функция, которая ожидает в аргументе объект str, обычно не будет принимать объект bytes и наоборот. По указанной причине Python 3.X по существу требует, чтобы вы придерживались одного типа или другого либо при необходимости выполняли ручные явные преобразования:

- str.encode() и bytes(S, encoding) транслируют строку в форму с низкоуровневыми байтами и в процессе создают закодированный объект bytes из декодированного объекта str;
- bytes.decode() и str(B, encoding) транслируют низкоуровневые байты в форму строки и в процессе создают декодированный объект str из закодированного объекта bytes.

Оба метода, encode и decode, а также вызовы open, которые нам предстоит рассмотреть, используют либо явно передаваемое имя кодировки, либо принятое по умолчанию. В Python 3.X стандартной кодировкой для методов всегда будет UTF-8, но open применяет значение из модуля locale, которое может варьироваться в зависимости от платформы. В Python 2.X стандартной кодировкой в обоих случаях обычно является ASCII, как отражено в модуле sys (что допускает изменения при начальном запуске). Вот пример для Python 3.X:

```
>>> S = 'eggs'
>>> S.encode()     # str->bytes: закодированный текст в низкоуровневые байты
b'eggs'
>>> bytes(S, encoding='ascii')    # str->bytes, альтернатива
b'eggs'

>>> B = b'spam'
>>> B.decode()      # bytes->str: декодированные низкоуровневые байты в текст
'spam'
>>> str(B, encoding='ascii')      # bytes->str, альтернатива
'spam'
```

Есть два предостережения. Во-первых, разнообразные стандартные кодировки вашей платформы доступны в модулях sys и locale, но аргумент кодировки в bytes обязателен, несмотря на необязательность в str.encode (и bytes.decode).

Во-вторых, хотя вызовы str не требуют аргумента кодировки, как делает bytes, его отсутствие в вызовах str вовсе не означает, что будет использовать стандартная кодировка — вызов str без аргумента кодировки возвращает выводимую строку объекта bytes, а не его преобразованную в str форму (обычно это не то, что вас будет интересовать!). Предположим, что B и S остались в том же виде, как в предыдущем взаимодействии:

```
>>> import sys, locale              # open() в Windows использует cp1252
                                    #   (подмножество Latin-1)
>>> sys.platform  # Но str() никогда не использует стандартную кодировку...
'win32'
>>> locale.getpreferredencoding(False), sys.getdefaultencoding()
('cp1252', 'utf-8')

>>> bytes(S)
TypeError: string argument without an encoding
TypeError: строковый аргумент без кодировки

>>> str(B)                          # str без кодировки
"b'spam'"                           # Выводимая строка, не преобразование!
>>> len(str(B))
7
>>> len(str(B, encoding='ascii')) # Используйте кодировку для
                                    #   преобразования в форму str
4
```

При наличии сомнений передавайте в Python 3.X аргумент с именем кодировки, даже если преобразование может иметь стандартный вариант. В *Python 2.X* преобразования похожи, хотя поддержка смешивания строковых типов в выражениях Python 2.X делает преобразования необязательными для текста ASCII и для другой модели строковых типов названия инструментов отличаются. В Python 2.X преобразования происходят между закодированным объектом str и декодированным объектом

unicode, а не так, как в Python 3.X – между закодированным объектом bytes и декодированным объектом str:

```
>>> S, U = 'spam', u'eggs' # Инструменты преобразования строковых типов Python 2.X
>>> S, U
('spam', u'eggs')
>>> unicode(S), str(U)      # Python 2.X преобразует str->unicode, unicode->str
(u'spam', 'eggs')
>>> S.decode(), U.encode() # в сравнении с bytes->str, str->bytes в Python 3.X
(u'spam', 'eggs')
```

Написание строк Unicode

Кодирование и декодирование станут более значимыми, когда вы начнете иметь дело с текстом Unicode, отличающимся от ASCII. Для записи произвольных в строках символов Unicode, часть которых может даже не удастся набрать на клавиатуре, строковые литералы Python поддерживают шестнадцатеричные байтовые управляющие последовательности "\xNN" и управляющие последовательности Unicode вида "\uNNNN" и "\UNNNNNNNN". В управляющих последовательностях Unicode первая форма дает четыре шестнадцатеричные цифры для кодирования 2-байтовой (16-битной) кодовой точки символа, а вторая – восемь цифр для 4-байтовой (32-битной) кодовой точки. Байтовые строки поддерживают только шестнадцатеричные управляющие последовательности для закодированного текста и другие формы для данных, основанных на байтах.

Написание текста ASCII

Давайте проработаем несколько примеров, которые продемонстрируют основы написания текста. Как было показано, текст ASCII является простым типом Unicode, хранящимся в виде последовательности байтовых значений, которые представляют символы:

```
C:\code> C:\python37\python
>>> ord('X')                # 'X' имеет двоичное значение кодовой точки 88
                            #   в стандартной кодировке
88
>>> chr(88)                 # 88 обозначает символ 'X'
'X'
>>> S = 'XYZ'               # Строка Unicode текста ASCII
>>> S
'XYZ'
>>> len(S)                  # Длина в три символа
3
>>> [ord(c) for c in S]     # Три символа с целочисленными порядковыми значениями
[88, 89, 90]
```

Нормальный 7-битный текст ASCII такого рода представлен с помощью одного символа на байт в каждой схеме кодирования Unicode, описанной ранее в главе:

```
>>> S.encode('ascii')       # Значения 0..127 в 1 байте (7 бит) каждое
b'XYZ'
>>> S.encode('latin-1')     # Значения 0..255 в 1 байте (8 бит) каждое
b'XYZ'
>>> S.encode('utf-8')       # Значения 0..127 в 1 байте, 128..2047 в 2 байтах,
                            #   остальные в 3 или 4 байтах
b'XYZ'
```

Фактически объекты bytes, возвращаемые за счет кодирования текста ASCII подобным способом, представляют собой последовательность коротких целых чисел, которые по возможности выводятся как символы ASCII:

```
>>> S.encode('latin-1')
b'XYZ'
>>> S.encode('latin-1')[0]
88
>>> list(S.encode('latin-1'))
[88, 89, 90]
```

Написание текста, отличающегося от ASCII

Формально для написания отличных от ASCII символов мы можем применять:

- шестнадцатеричные управляющие последовательности или управляющие последовательности Unicode для внедрения порядковых значений кодовых точек в текстовые строки – нормальные строковые литералы в Python 3.X и строковые литералы Unicode в Python 2.X (а также в Python 3.3 и последующих версиях для совместимости);
- шестнадцатеричные управляющие последовательности для внедрения закодированного представления символов в байтовые строки – нормальные строковые литералы в Python 2.X и литералы в виде байтовых строк в Python 3.X (а также в Python 2.X для совместимости).

Обратите внимание, что текстовые строки содержат действительные значения кодовых точек, тогда как байтовые строки содержат их закодированную форму. Значение закодированного представления символа в байтовой строке будет таким же, как и значение его декодированной кодовой точки Unicode в текстовой строке, только для определенных символов и кодировок. В любом случае шестнадцатеричные управляющие последовательности ограничены указанием однобайтовых значений, но с помощью управляющих последовательностей Unicode можно указывать символы со значениями шириной 2 и 4 байта. Функция chr также может использоваться для создания одиночного символа не ASCII из значения кодовой точки и, как мы увидим позже, объявления в исходном коде применяются к таким символам, внедренным в сценарий.

Например, шестнадцатеричные значения 0xC4 и 0xE8 являются кодами для двух специальных диакритических знаков за пределами 7-битного диапазона ASCII, но мы можем внедрять их в объекты str в Python 3.X, потому что тип str поддерживает Unicode:

```
>>> chr(0xc4)                # 0xC4, 0xE8: символы за пределами диапазона ASCII
'Ä'
>>> chr(0xe8)
'è'

>>> S = '\xc4\xe8'           # 8-битные шестнадцатеричные управляющие
                             #   последовательности: две цифры
>>> S
'Äè'

>>> S = '\u00c4\u00e8'       # 16-битные управляющие последовательности
                             #   Unicode: четыре цифры каждое
>>> S
'Äè'
>>> len(S)                   # Длина в два символа (не количество байтов!)
2
```

Обратите внимание, что в строковых литералах с текстом Unicode вроде показанных выше шестнадцатеричные управляющие последовательности и управляющие последовательности Unicode указывают значение кодовой точки Unicode, но не байтовое значение. Шестнадцатеричная управляющая последовательность x требует в точности две цифры (для 8-битных значений кодовых точек), а управляющая последовательность Unicode с u и U — соответственно четыре и восемь шестнадцатеричных цифр для указания значений кодовых точек, которые могут достигать 16 и 32 битов:

```
>>> S = '\U000000c4\U000000e8' # 32-битные управляющие последовательности
                                #  Unicode: восемь цифр каждое
>>> S
'Äè'
```

Как будет показано позже, в данном отношении Python 2.X работает аналогично, но управляющие последовательности Unicode разрешены только в литеральной форме Unicode. Они работают здесь в нормальных строковых литералах Python 3.X лишь потому, что нормальные строки в этой линейке всегда представлены в кодировке Unicode.

Кодирование и декодирование текста, отличающегося от ASCII

Если теперь мы попытаемся *закодировать* строку с текстом, отличающимся от ASCII, из предыдущего раздела в низкоуровневые байты как *ASCII*, то получим ошибку, потому что ее символы выходят за пределы диапазона 7-битных значений кодовых точек ASCII:

```
>>> S = '\u00c4\u00e8'    # Строка с текстом не ASCII длиной в два символа
>>> S
'Äè'
>>> len(S)
2

>>> S.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)
Ошибка кодирования Unicode: кодек ascii не может закодировать символы в
позициях 0-1: порядковое значение не в диапазоне range(128)
```

Однако кодирование такой текстовой строки как *Latin-1* работает, поскольку каждый символ попадает в 8-битный диапазон данной кодировки, и мы получаем 1 байт на символ, выделенный в закодированной байтовой строке. Кодирование как *UTF-8* тоже работает: эта кодировка поддерживает широкий диапазон кодовых точек Unicode, но взамен выделяет 2 байта на символ, отличный от ASCII. Если такие закодированные строки записываются в файл, тогда показанные здесь в качестве результатов кодирования низкоуровневые объекты bytes и будут тем, что фактически сохраняется в файле для указанных типов кодировок:

```
>>> S.encode('latin-1')         # 1 байт на символ при кодировании
b'\xc4\xe8'
>>> S.encode('utf-8')           # 2 байта на символ при кодировании
b'\xc3\x84\xc3\xa8'
>>> len(S.encode('latin-1'))  # 2 байта в latin-1, 4 байта в utf-8
2
>>> len(S.encode('utf-8'))
4
```

Обратите внимание, что вы также можете пойти другим путем, *читая* низкоуровневые байты из файла и *декодируя* их обратно в строку Unicode. Тем не менее, как будет объясняться позже, задаваемый в вызове open режим кодирования приводит к автоматическому выполнению соответствующего декодирования при вводе (и позволяет избежать проблем, которые могут возникнуть из-за чтения неполных последовательностей символов, когда производится чтение блоками байтов):

```
>>> B = b'\xc4\xe8'         # Текст, закодированный согласно Latin-1
>>> B
b'\xc4\xe8'
>>> len(B)                  # 2 низкоуровневых байта, 2 закодированных символа
2
>>> B.decode('latin-1')     # Декодирование в текст согласно Latin-1
'Äè'

>>> B = b'\xc3\x84\xc3\xa8' # Текст, закодированный согласно UTF-8
>>> len(B)                  # 4 низкоуровневых байта, 2 закодированных символа
4
>>> B.decode('utf-8')       # Декодирование в текст согласно UTF-8
'Äè'
>>> len(B.decode('utf-8'))  # Два символа Unicode в памяти
2
```

Другие схемы кодирования

Некоторые кодировки используют для представления символов даже более длинные последовательности байтов. При необходимости вы можете указывать для символов в своих строках 16- и 32-битные значения кодовых точек Unicode. Как было показано ранее, в первом случае мы можем применять "\u..." с четырьмя шестнадцатеричными цифрами, а во втором — "\U..." с восемью шестнадцатеричными цифрами, и свободно смешивать их в литералах с более простыми символами ASCII:

```
>>> S = 'A\u00c4B\U000000e8C'
>>> S                            # A, B, C и 2 символа не ASCII
'AÄBèC'
>>> len(S)                       # Длина в пять символов
5

>>> S.encode('latin-1')
b'A\xc4B\xe8C'
>>> len(S.encode('latin-1'))     # 5 байтов при кодировании согласно latin-1
5

>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> len(S.encode('utf-8'))       # 7 байтов при кодировании согласно utf-8
7
```

Формально говоря, вы также можете строить строки Unicode по частям, используя вместо управляющих последовательностей Unicode или шестнадцатеричных управляющих последовательностей встроенную функцию chr, но для крупных строк это может стать утомительным:

```
>>> S = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'
>>> S
'AÄBèC'
```

Однако другие кодировки могут применять совершенно другие байтовые форматы. Например, кодировка EBCDIC (cp500) даже ASCII не кодирует так, как кодировки, используемые нами до сих пор; поскольку кодирование и декодирование выполняется интерпретатором Python, этот момент нас будет интересовать обычно лишь при предоставлении имен кодировок для источников данных:

```
>>> S
'AÄBèC'
>>> S.encode('cp500')     # Другие две западноевропейские кодировки
b'\xc1c\xc2T\xc3'
>>> S.encode('cp850')     # 5 байтов каждая, разные закодированные значения
b'A\x8eB\x8aC'

>>> S = 'spam'    # В большинстве кодировок текст ASCII остается тем же самым
>>> S.encode('latin-1')
b'spam'
>>> S.encode('utf-8')
b'spam'
>>> S.encode('cp500')     # Но не в cp500: IBM EBCDIC!
b'\xa2\x97\x81\x94'
>>> S.encode('cp850')
b'spam'
```

То же самое остается справедливым для кодировок UTF-16 и UTF-32, которые применяют схемы с фиксированными 2 и 4 байтами на символ и заголовками одинаковых размеров – отличающиеся от ASCII символы кодируются по-другому, а символы ASCII занимают не один байт:

```
>>> S = 'A\u00c4B\U000000e8C'
>>> S.encode('utf-16')
b'\xff\xfeA\x00\xc4\x00B\x00\xe8\x00C\x00'

>>> S = 'spam'
>>> S.encode('utf-16')
b'\xff\xfes\x00p\x00a\x00m\x00'
>>> S.encode('utf-32')
b'\xff\xfe\x00\x00s\x00\x00\x00p\x00\x00\x00a\x00\x00\x00m\x00\x00\x00'
```

Байтовые строковые литералы: закодированный текст

Здесь также уместно сделать два предостережения. Во-первых, Python 3.X позволяет записывать специальные символы в строках str с помощью шестнадцатеричных управляющих последовательностей и управляющих последовательностей Unicode – в литералах bytes последние принимаются в буквальной форме, не как управляющие последовательности. На самом деле для надлежащего вывода символов не ASCII строки bytes должны быть декодированы в строки str:

```
>>> S = 'A\xC4B\xE8C'  # Python 3.X: str распознает шестнадцатеричные управляющие
>>> S             # последовательности и управляющие последовательности Unicode
'AÄBèC'
>>> S = 'A\u00C4B\U000000E8C'
>>> S
'AÄBèC'

>>> B = b'A\xC4B\xE8C'     # bytes распознает шестнадцатеричные управляющие
          # последовательности, но не управляющие последовательности Unicode
b'A\xc4B\xe8C'
```

```
>>> B = b'A\u00C4B\U000000E8C'       # Управляющие последовательности
                                     #   принимаются в буквальной форме!
>>> B
b'A\\u00C4B\\U000000E8C'
>>> B = b'A\xC4B\xE8C'               # Использование шестнадцатеричных управляющих
                                     #   последовательностей для bytes
>>> B                                # Отличающиеся от ASCII символы выводятся
                                     #   как шестнадцатеричные цифры
b'A\xc4B\xe8C'
>>> print(B)
b'A\xc4B\xe8C'
>>> B.decode('latin-1')              # Декодирование согласно latin-1 для
                                     #   интерпретации как текста
'AÄBèC'
```

Во-вторых, литералы bytes требуют либо символов ASCII, либо управляющих последовательностей, если значения превышают 127. С другой стороны, строки str допускают литералы, содержащие любой символ из исходного набора символов, которым, как обсуждалось ранее, по умолчанию будет UTF-8 в Python 3.X (и ASCII в Python 2.X), если только в исходном файле не указано объявление кодировки:

```
>>> S = 'AÄBèC'                      # Символы из UTF-8, если отсутствует
                                     #   объявление кодировки
>>> S
'AÄBèC'
>>> B = b'AÄBèC'
SyntaxError: bytes can only contain ASCII literal characters.
Синтаксическая ошибка: строка bytes может содержать только литеральные
символы ASCII.
>>> B = b'A\xC4B\xE8C'               # Символы должны быть ASCII или управляющими
                                     #   последовательностями
>>> B
b'A\xc4B\xe8C'
>>> B.decode('latin-1')
'AÄBèC'
>>> S.encode()              # По умолчанию исходный код кодируется согласно UTF-8
b'A\xc3\x84B\xc3\xa8C'               # Используется стандартная кодировка системы,
                                     #   если она не передана явно
>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> B.decode()                       # Низкоуровневые байты не соответствуют utf-8
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: ...
Ошибка декодирования Unicode: кодек utf8 не может декодировать байты в
позициях 1-2: ...
```

Оба ограничения обретут смысл, когда вы вспомните, что байтовые строки хранят данные, основанные на байтах, не декодированные порядковые значения кодовых точек Unicode. Наряду с тем, что они могут содержать закодированную форму текста, декодированные значения кодовых точек не вполне применимы к байтовым строкам, если только символы предварительно не закодированы.

Преобразования между кодировками

До сих пор мы кодировали и декодировали строки для инспектирования их структуры. Строку можно также преобразовывать в кодировку, отличающуюся от первоначальной, но потребуется предоставить явное имя кодировки для кодирования и декодирования. Так поступать необходимо вне зависимости от того, откуда поступает исходная текстовая строка – из файла либо из литерала.

Термин *преобразование* может здесь использоваться не совсем правильно – в действительности он означает всего лишь кодирование текстовой строки в низкоуровневые байты по другой схеме, а не той, из которой производилось декодирование. Как подчеркивалось ранее, декодированный текст в памяти не имеет типа кодировки и является просто строкой из кодовых точек Unicode (они же символы); концепция изменения его кодировки в такой форме отсутствует. Тем не менее, эта схема позволяет сценариям читать данные в одной кодировке и сохранять их в другой, чтобы поддерживать множество клиентов для тех же самых данных:

```
>>> B = b'A\xc3\x84B\xc3\xa8C # Текст, первоначально закодированный в
                               #   формате UTF-8
>>> S = B.decode('utf-8')      # Декодирование в текст Unicode согласно UTF-8
>>> S
'AÄBèC'
>>> T = S.encode('cp500')      # Преобразование в закодированную строку bytes
                               #   согласно EBCDIC
>>> T
b'\xc1c\xc2T\xc3'
>>> U = T.decode('cp500')      # Преобразование обратно в Unicode согласно EBCDIC
>>> U
'AÄBèC'
>>> U.encode()                 # Снова согласно стандартной кодировке utf-8
b'A\xc3\x84B\xc3\xa8C'
```

Имейте в виду, что специальные управляющие последовательности Unicode и шестнадцатеричные управляющие последовательности необходимы только при ручном написании строк Unicode, отличающихся от ASCII. На практике вы будете часто загружать такой текст из файлов. Как будет показано далее в главе, файловый объект Python 3.X (создаваемый посредством встроенной функции `open`) автоматически декодирует текстовые строки при чтении и кодирует их при записи; по этой причине ваш сценарий нередко может иметь дело со строками обобщенным образом, не представляя специальные символы напрямую.

Позже в главе вы также увидите, что преобразования между кодировками можно осуществлять во время передачи строк в файлы и из файлов, применяя методику, которая очень похожа на используемую в последнем примере. Хотя при открытии файла вам все равно придется указывать явные имена кодировок, файловый интерфейс выполняет б*о*льшую часть работы по преобразованию автоматически.

Кодирование строк Unicode в Python 2.X

В настоящей главе я делаю акцент на поддержке Unicode в Python 3.X, поскольку она относительно нова. Но теперь, когда были изложены основы строк Unicode в Python 3.X, я должен более полно объяснить, как добиться многого того же в Python 2.X, хотя инструменты и отличаются. Тип `unicode` доступен в Python 2.X, но является отдельным от `str`, поддерживает большинство тех же самых операций и разрешает смешивание нормальных строк и строк Unicode, когда объект `str` содержит все символы ASCII.

Фактически вы можете считать str из Python 2.X типом bytes из Python 3.X, когда дело доходит до декодирования низкоуровневых байтов в строку Unicode, при условии, что они имеют надлежащую форму. Ниже приведен интерактивный сеанс Python 2.X; в Python 2.X символы Unicode отображаются в шестнадцатеричном виде, если только не выводятся явно, а отображение символов не ASCII может варьироваться в зависимости от оболочки (большинство кода в текущем разделе запускалось вне IDE-среды IDLE, которая временами обнаруживает и выводит символы Latin-1 в закодированных байтовых строках — позже будет описана переменная среды PYTHONIOENCODING и проблемы отображения в окне командной строки Windows):

```
C:\code> C:\python27\python
>>> S = 'A\xC4B\xE8C'      # Строка 8-битных байтов
>>> S                      # Текст, закодированный согласно Latin-1,
                           #   некоторые символы не ASCII
'A\xc4B\xe8C'
>>> print S                # Непечатаемые символы (в IDLE может быть по-другому)
A▣B▣C

>>> U = S.decode('latin1') # Декодирование bytes в текст Unicode согласно latin-1
>>> U
u'A\xc4B\xe8C'
>>> print U
AÄBèC

>>> S.decode('utf-8')      # Закодированная форма не совместима с utf-8
UnicodeDecodeError: 'utf8' codec can't decode byte 0xc4 in position 1:
invalid continuation byte
Ошибка декодирования Unicode: кодек utf8 не может декодировать байт 0xc4 в
позиции 1: недопустимый байт продолжения

>>> S.decode('ascii')      # Закодированные байты также выходят за пределы
                           #   диапазона ASCII
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1:
ordinal not in range(128)
Ошибка декодирования Unicode: кодек ascii не может декодировать байт 0xc4 в
позиции 1: порядковое значение не в диапазоне range(128)
```

Для написания текста Unicode создайте объект unicode с помощью литеральной формы u'xxx' (как уже упоминалось, начиная с версии Python 3.3, он снова стал доступным, но в целом избыточен в Python 3.X, т.к. нормальные строки поддерживают Unicode):

```
>>> U = u'A\xC4B\xE8C'     # Создать строку Unicode, шестнадцатеричные
                           #   управляющие последовательности
>>> U
u'A\xc4B\xe8C'

>>> print U
'AÄBèC'
```

После создания объекта unicode вы можете преобразовывать текст Unicode в другие кодировки с низкоуровневыми байтами подобно кодированию объектов str в объекты bytes в Python 3.X:

```
>>> U.encode('latin-1')    # Кодирование согласно latin-1: 8-битные байты
'A\xc4B\xe8C'

>>> U.encode('utf-8')      # Кодирование согласно utf-8: многобайтовая строка
'A\xc3\x84B\xc3\xa8C'
```

В точности как в Python 3.X, в Python 2.X символы не ASCII могут записываться посредством шестнадцатеричных управляющих последовательностей или управляющих последовательностей Unicode. Однако, как и в случае типа bytes из Python 3.X, управляющие последовательности "\u..." и "\U..." в Python 2.X распознаются только для строк unicode, но не 8-битных строк str — к тому же они используются для предоставления значений декодированных порядковых целых чисел Unicode, которые не имеют смысла в строке с низкоуровневыми байтами:

```
C:\code> C:\python27\python
>>> U = u'A\xC4B\xE8C'              # Шестнадцатеричные управляющие
                                    #   последовательности для символов не ASCII
>>> U
u'A\xc4B\xe8C'
>>> print U
AÄBèC

>>> U = u'A\u00C4B\U000000E8C'      # Управляющие последовательности Unicode
                                    #   для символов не ASCII
>>> U                               # u'' = 16 бит, U'' = 32 бита
u'A\xc4B\xe8C'
>>> print U
AÄBèC

>>> S = 'A\xC4B\xE8C'
# Шестнадцатеричные управляющие последовательности работают
>>> S
'A\xc4B\xe8C'
>>> print S                         # Но некоторые могут выводится странно,
                                    #   не будучи декодированными
A□B□C
>>> print S.decode('latin-1')
AÄBèC

>>> S = 'A\u00C4B\U000000E8C'       # Не управляющие последовательности Unicode:
                                    #   берутся буквально!
>>> S
'A\\u00C4B\\U000000E8C'
>>> print S
A\u00C4B\U000000E8C
>>> len(S)
19
```

Смешивание строковых типов в Python 2.X

Подобно типам str и bytes из Python 3.X типы unicode и str из Python 2.X разделяют почти идентичные наборы операций, так что при отсутствии необходимости преобразования в другие кодировки вы часто можете трактовать тип unicode так, как если бы он был str. Тем не менее, одно из основных отличий между линейками Python 2.X и Python 3.X заключается в том, что объекты unicode и отличающиеся от Unicode объекты str можно свободно *смешивать* в выражениях Python 2.X — до тех пор, пока объект str совместим с объектом unicode, интерпретатор Python будет автоматически преобразовывать его в unicode:

```
>>> u'ab' + 'cd'        # В случае совместимости в Python 2.X можно смешивать
u'abcd'                 # Но 'ab' + b'cd' в Python 3.X не разрешено
```

Однако такой либеральный подход к смешиванию строковых типов в Python 2.X работает, *только* если 8-битные строки содержат исключительно 7-битные (ASCII) байты:

```
>>> S = 'A\xC4B\xE8C'           # Нельзя смешивать в Python 2.X,
                                #   если str содержит символы не ASCII!
>>> U = u'A\xC4B\xE8C'
>>> S + U
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1:
ordinal not in range(128)
Ошибка декодирования Unicode: кодек ascii не может декодировать байт 0xc4 в
позиции 1: порядковое значение не в диапазоне range(128)
>>> 'abc' + U                   # Можно смешивать, только если str содержит
                                #   все 7-битные символы ASCII
u'abcA\xc4B\xe8C'
>>> print 'abc' + U             # Использовать print для отображения символов
abcAÄBèC
>>> S.decode('latin-1') + U     # В Python 2.X также может потребоваться
                                #   ручное преобразование
u'A\xc4B\xe8CA\xc4B\xe8C'
>>> print S.decode('latin-1') + U
AÄBèCAÄBèC
>>> print u'\xA3' + '999.99'    # См. также пример с денежными значениями
                                #   из главы 25 первого тома
£999.99
```

Напротив, в Python 3.X типы `str` и `bytes` *никогда* автоматически не смешиваются и требуют ручных преобразований – фактически предыдущий код выполняется в Python 3.3 и последующих версиях, но лишь потому, что интерпретатор Python 3.X считает литерал Unicode из Python 2.X таким же, как нормальная строка (`u` игнорируется); эквивалентом Python 3.X могло бы быть сложение объектов `str` и `bytes` (т.е. `'ab'+b'cd'`), что потерпит неудачу в Python 3.X, если не преобразовать объекты в общий тип.

Тем не менее, в Python 2.X несходство типов часто несущественно для кода. Подобно нормальным строкам строки Unicode можно сцеплять, индексировать, нарезать, сопоставлять с образцом посредством модуля `re` и т.д., к тому же их нельзя изменять на месте. Если вам когда-либо понадобится явно выполнять преобразования между двумя типами, тогда можете применять встроенные функции `str` и `unicode`:

```
>>> str(u'spam')                # Unicode в нормальную
'spam'
>>> unicode('spam')             # Нормальная в Unicode
u'spam'
```

Если вы используете Python 2.X, тогда просмотрите пример другого файлового интерфейса, приводимый позже в главе. Вызов `open` поддерживает только файлы 8-битных байтов, возвращая их содержимое в виде строк `str`, которое вы можете по своему усмотрению интерпретировать как текстовые или двоичные данные и при необходимости декодировать его. Для чтения и записи файлов Unicode и автоматического кодирования либо декодирования их содержимого применяйте в Python 2.X вызов `codecs.open`, который вы увидите в действии далее в главе. Упомянутый вызов обеспечивает почти такую же функциональность, как `open` в Python 3.X, и для представления содержимого файла использует объекты `unicode` из Python 2.X. При чтении файла закодированные байты транслируются в декодированные символы Unicode, а при записи строки переводятся в желаемую кодировку, которая была указана во время открытия файла.

Объявления кодировок в файлах исходного кода

Наконец, управляющие последовательности Unicode хорошо подходят для редких символов Unicode в строковых литералах, но могут стать утомительными, когда внедрять в строки отличающийся от ASCII текст приходится часто. Чтобы интерпретировать содержимое строк, которые вы записываете и, следовательно, внедряете внутрь текста своих файлов сценариев, в качестве стандартной кодировки Python применяет UTF-8 в линейке Python 3.X (и ASCII в линейке Python 2.X). Однако Python Python позволяет использовать произвольные кодировки и поддерживаемые ими наборы символов за счет включения комментария, в котором указывается желаемая кодировка. Комментарий обычно имеет показанную ниже форму и должен находиться в первой или во второй строке сценария в Python 2.X или 3.X:

```
# -*- coding: latin-1 -*-
```

При наличии комментария такого вида интерпретатор Python будет распознавать строки, изначально представленные в заданной кодировке. Это означает, что вы можете редактировать свой файл сценария в текстовом редакторе, который корректно воспринимает и отображает диакритические и другие символы не ASCII, а интерпретатор Python будет правильно их декодировать в строковых литералах. Например, взгляните, как комментарий в начале файла `text.py` с представленным далее содержимым позволяет встраивать символы Latin-1 в строки, которые сами встраиваются в текст файла сценария:

```
# -*- coding: latin-1 -*-
# Любая из следующих форм литеральных строк работает в latin-1.
# Изменение кодировки выше на либо ascii, либо utf-8 приведет к неудаче,
# потому что тогда 0xc4 и 0xe8 в myStr1 не будут допустимыми.

myStr1 = 'aÄBèC'

myStr2 = 'A\u00c4B\U000000e8C'

myStr3 = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'

import sys
print('Default encoding:', sys.getdefaultencoding())  # Стандартная кодировка

for aStr in myStr1, myStr2, myStr3:
    print('{0}, strlen={1}, '.format(aStr, len(aStr)), end='')

    bytes1 = aStr.encode()              # Согласно стандартной кодировке utf-8:
                                        #   2 байта для не ASCII
    bytes2 = aStr.encode('latin-1')     # Один байт на символ
    # bytes3 = aStr.encode('ascii')     # ASCII потерпит неудачу:
                                        #   за пределами диапазона 0..127

    print('byteslen1={0}, byteslen2={1}'.format(len(bytes1), len(bytes2)))
```

Запуск сценария приводит к выдаче следующего вывода, где для каждой из трех методик кодирования отображается строка, ее длина и длины их форм байтовых строк, закодированных согласно UTF-8 и Latin-1:

```
C:\code> C:\python37\python text.py
Default encoding: utf-8
aÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
```

Поскольку многие программисты, вероятно, будут придерживаться стандартных кодировок исходного кода, я предлагаю поискать дополнительные сведения об этом варианте и о других расширенных темах поддержки Unicode в наборе руководств по Python. Здесь же мы кратко рассмотрим новые типы объектов байтовых строк в Python 3.X и затем перейдем к исследованию изменений файлов и инструментов.

Добавочные примеры написания символов не ASCII и объявлений в файлах исходного кода можно найти в сценарии форматировании денежных значений из главы 25 первого тома и в файле `formats_currency2.py`, входящем в состав загружаемого кода для настоящей книги. Последний требует объявления в файле исходного кода, чтобы с ним мог работать интерпретатор Python, т.к. он содержит символы валют, отличающиеся от ASCII. Данный пример также иллюстрирует улучшение переносимости, возможное в случае применения литерала Unicode из Python 2.X в коде Python 3.X, начиная с версии Python 3.3.

Использование объектов `bytes` в Python 3.X

В главе 7 первого тома мы исследовали широкий выбор операций, доступных для универсального строкового типа `str` в Python 3.X; базовый строковый тип в линейках Python 2.X и Python 3.X работает идентично, поэтому мы не будем возвращаться к данной теме. Взамен давайте чуть глубже проанализируем набор операций, предлагаемый новым типом `bytes` в Python 3.X.

Как упоминалось ранее, объект `bytes` в Python 3.X представляет собой последовательность коротких целых чисел, каждое из которых находится в диапазоне 0–255 и потому отображается как символ ASCII. Он поддерживает операции над последовательностями и большинство методов, доступных в объектах `str` (и присутствующих в типе `str` из Python 2.X). Тем не менее, `bytes` *не* поддерживает метод `format` или выражение форматирования `%`, к тому же смешивать объекты типов `bytes` и `str` не допускается без явных преобразований. Обычно вы будете использовать все объекты типа `str` и текстовые файлы для *текстовых данных*, а все объекты типа `bytes` и двоичные файлы для *двоичных данных*.

Вызовы методов

Если вы действительно хотите увидеть, какие атрибуты имеет тип `str`, которые отсутствуют в `bytes`, то всегда можете проверить результаты вызова встроенной функции `dir`. Вывод также может сообщить кое что о поддерживаемых ими операциях выражений (например, `__mod__` и `__rmod__` реализуют операцию `%`):

```
C:\code> C:\python37\python

# Атрибуты, имеющиеся в str, но не в bytes
>>> set(dir('abc')) - set(dir(b'abc'))
{'isprintable', 'format', 'isdecimal', 'encode', 'format_map',
'casefold', 'isidentifier', 'isnumeric'}

# Атрибуты, имеющиеся в bytes, но не в str
>>> set(dir(b'abc')) - set(dir('abc'))
{'fromhex', 'decode', 'hex'}
```

Как видите, функциональность типов `str` и `bytes` почти идентична. Их уникальные атрибуты, как правило, являются методами, которые применимы к одному типу, но неприменимы к другому; скажем, `decode` транслирует низкоуровневый объ-

ект bytes в его представление str, а encode переводит строку в ее низкоуровневое представление bytes. Большинство методов одинаковы, хотя методы bytes требуют аргументов bytes (опять-таки строковые типы Python 3.X не смешиваются). Также вспомните, что объекты bytes неизменяемы как и объекты str в Python 2.X и 3.X (для краткости сообщения об ошибках далее показаны не полностью):

```
>>> B = b'spam'                # Литерал b'...' типа bytes
>>> B.find(b'pa')
1

>>> B.replace(b'pa', b'XY')    # Методы bytes ожидают аргументов bytes
b'sXYm'

>>> B.split(b'pa')             # Методы bytes возвращают результаты bytes
[b's', b'm']

>>> B
b'spam'
>>> B[0] = 'x'
TypeError: 'bytes' object does not support item assignment
Ошибка типа: объект bytes не поддерживает присваивание в отношении элементов
```

Одно заметное отличие заключается в том, что *строковое форматирование* в Python 3.X работает только с объектами str, но не с объектами bytes (выражения и методы строкового форматирования обсуждались в главе 7 первого тома):

```
>>> '%s' % 99
'99'
>>> b'%s' % 99
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'
Ошибка типа: неподдерживаемые типы операндов для %: bytes и int

>>> '{0}'.format(99)
'99'
>>> b'{0}'.format(99)
AttributeError: 'bytes' object has no attribute 'format'
Ошибка атрибута: объект bytes не имеет атрибута format
```

Операции над последовательностями

Помимо вызовов методов все обычные универсальные операции над последовательностями, которые вы знаете (и возможно предпочитаете) по строкам и спискам Python 2.X, работают ожидаемым образом для типов str и bytes в Python 3.X; к ним относятся индексирование, нарезание, конкатенация и т.д. В показанном ниже взаимодействии обратите внимание, что индексирование объекта bytes возвращает целое число, дающее двоичное значение байта; на самом деле bytes представляет собой *последовательность 8-битных целых чисел*, которая при отображении целиком для удобства выводится как строка символов ASCII, когда это возможно. Чтобы проверить значение заданного байта, используйте встроенную функцию chr для его преобразования в соответствующий символ:

```
>>> B = b'spam'       # Последовательность коротких целых чисел
>>> B                 # Выводится как символы ASCII (и/или шестнадцатеричные
                      #   управляющие последовательности)
b'spam'

>>> B[0]              # Индексирование выдает целое число
115
>>> B[-1]
```

```
109
>>> chr(B[0])              # Отображает символ для целого числа
's'
>>> list(B)                # Отображает целые значения всех байтов
[115, 112, 97, 109]
>>> B[1:], B[:-1]
(b'pam', b'spa')
>>> len(B)
4
>>> B + b'lmn'
b'spamlmn'
>>> B * 4
b'spamspamspamspam'
```

Другие способы создания объектов bytes

До сих пор мы создавали объекты bytes главным образом с помощью синтаксиса литералов b'...'. Мы также можем создавать их вызовом конструктора bytes с объектом str и именем кодировки, вызовом конструктора bytes с итерируемым объектом целых чисел, представляющих значения байтов, или кодированием объекта str согласно стандартной (или переданной) кодировки. Как мы видели, операция кодирования берет текстовый объект str и возвращает низкоуровневые значения закодированных байтов строки в соответствии с указанной кодировкой; и наоборот, операция декодирования берет низкоуровневую последовательность bytes и транслирует ее в представление текстовой строки str – последовательность символов Unicode. Обе операции создают новые строковые объекты:

```
>>> B = b'abc'                    # Литерал
>>> B
b'abc'

>>> B = bytes('abc', 'ascii')  # Конструктор с именем кодировки
>>> B
b'abc'

>>> ord('a')
97
>>> B = bytes([97, 98, 99])     # Итерируемый объект с целыми числами
>>> B
b'abc'

>>> B = 'spam'.encode()          # str.encode() (или bytes())
>>> B
b'spam'
>>>
>>> S = B.decode()               # bytes.decode() (или str())
>>> S
'spam'
```

С функциональной точки зрения последние две операции из показанных выше в действительности являются инструментами для *преобразований* между str и bytes, о которых шла речь ранее и которые более подробно рассматриваются в следующем разделе.

Смешивание строковых типов

В вызов replace из раздела "Вызовы методов" ранее в главе мы должны были передать два объекта bytes — типы str в нем не работают. Несмотря на то что в Python 2.X автоматически выполняются преобразования str в и из unicode, когда они возможны (т.е. когда str содержит 7-битный текст ASCII), Python 3.X в ряде контекстов требует определенных строковых типов и ожидает ручных преобразований, если они необходимы:

```
# Вызовам методов и функций должны передаваться ожидаемые типы

>>> B = b'spam'

>>> B.replace('pa', 'XY')
TypeError: expected an object with the buffer interface
Ошибка типа: ожидался объект с интерфейсом буфера

>>> B.replace(b'pa', b'XY')
b'sXYm'

>>> B = B'spam'
>>> B.replace(bytes('pa'), bytes('xy'))
TypeError: string argument without an encoding
Ошибка типа: строковый аргумент без кодировки

>>> B.replace(bytes('pa', 'ascii'), bytes('xy', 'utf-8'))
b'sxym'

# В выражениях со смешанными типами Python 3.X должны выполняться
#   ручные преобразования

>>> b'ab' + 'cd'
TypeError: can't concat bytes to str
Ошибка типа: не удалось выполнить конкатенацию bytes и str

>>> b'ab'.decode() + 'cd'          # bytes в str
'abcd'
>>> b'ab' + 'cd'.encode()          # str в bytes
b'abcd'
>>> b'ab' + bytes('cd', 'ascii')   # str в bytes
b'abcd'
```

Хотя вы можете самостоятельно создавать объекты bytes для представления упакованных двоичных данных, они также могут порождаться автоматически при чтении файлов, открытых в двоичном режиме, как будет показано далее в главе. Но сначала давайте займемся родственным bytes типом, допускающим изменения на месте.

Использование объектов bytearray в Python 3.X/2.6+

До сих пор наше внимание было сосредоточено на типах str и bytes, потому что они соответствуют типам unicode и str из Python 2. Однако в Python 3.X появился третий строковый тип, bytearray — изменяемая последовательность целых чисел в диапазоне от 0 до 255, — который является изменяемым вариантом типа bytes. Как таковой, он поддерживает те же самые строковые методы и операции над последовательностями, что и bytes, а также многие операции изменения на месте, поддерживаемые *списками*.

Байтовые массивы поддерживают изменения на месте как по-настоящему двоичных данных, так и простых форм текста вроде ASCII, которые могут быть представлены с помощью 1 байта на символ (обогащенный текст Unicode по-прежнему требует неизменяемых строк Unicode). Тип bytearray также доступен в Python 2.6 и 2.7 в качестве переноса из Python 3.X, но не обеспечивает строгого различения между текстовыми и двоичными данными, как в Python 3.X.

Объекты bytearray в действии

Давайте проведем краткий тур. Мы можем создавать объекты bytearray вызовом встроенной функции bytearray. В Python 2.X для инициализации может применяться любая строка:

```
# Создание в Python 2.6/2.7: изменяемая последовательность коротких
#  целых чисел (0..255)

>>> S = 'spam'
>>> C = bytearray(S)              # Перенос из Python 3.X в Python 2.6+
>>> C                             # b'..' == '..' в Python 2.6+ (str)
bytearray(b'spam')
```

В Python 3.X имя кодировки или байтовая строка обязательны, т.к. текстовые и двоичные строки не смешиваются (хотя байтовые строки могут отражать закодированный текст Unicode):

```
# Создание в Python 3.X: текстовые и двоичные строки не смешиваются

>>> S = 'spam'
>>> C = bytearray(S)
TypeError: string argument without an encoding
Ошибка типа: строковый аргумент без кодировки

>>> C = bytearray(S, 'latin1')  # Тип, специфичный к содержимому, в Python 3.X
>>> C
bytearray(b'spam')

>>> B = b'spam'                   # b'..' != '..' в Python 3.X (bytes/str)
>>> C = bytearray(B)
>>> C
bytearray(b'spam')
```

Созданные объекты bytearray представляют собой последовательности коротких целых чисел вроде bytes и являются изменяемыми подобно спискам, хотя присваивание по индексу требует применения целого числа, а не строки (если не указано иное, то все последующие примеры будут продолжением текущего сеанса под управлением Python 3.X):

```
# Изменяемый объект, но присваивание по индексу требует указания
#  целого числа, а не строки

>>> C[0]
115

>>> C[0] = 'x'           # Это и следующее выражение работает в Python 2.6/2.7
TypeError: an integer is required
Ошибка типа: требуется целое число
>>> C[0] = b'x'
TypeError: an integer is required
Ошибка типа: требуется целое число
```

```
>>> C[0] = ord('x')              # Использование ord() для получения
                                 #   порядкового значения символа
>>> C
bytearray(b'xpam')

>>> C[1] = b'Y'[0]               # Либо индексирование байтовой строки
>>> C
bytearray(b'xYam')
```

Обработка объектов bytearray позаимствована от строк и списков, т.к. они представляют собой изменяемые байтовые строки. Наряду с тем, что методы типа byterrray перекрываются с методами str и bytes, в нем также есть много методов list. Помимо именованных методов методы __iadd__ и __setitem__ в bytearray реализуют соответственно конкатенацию на месте += и присваивание по индексу:

```
# B bytes, но не в bytearray
>>> set(dir(b'abc')) - set(dir(bytearray(b'abc')))
{'__getnewargs__'}

# B bytearray, но не в bytes
>>> set(dir(bytearray(b'abc'))) - set(dir(b'abc'))
{'__alloc__', '__setitem__', 'remove', '__delitem__', 'extend', 'clear',
'insert', 'reverse', 'pop', 'copy', '__imul__', 'append', '__iadd__'}
```

Вы можете изменять объект bytearray на месте посредством присваивания по индексу, как только что было показано, и методов, похожих на списковые методы, вроде приведенных ниже (чтобы изменить текст на месте в версиях, предшествующих Python 2.6, его пришлось бы преобразовать в список и затем обратно с помощью list(str) и ''.join(list) — примеры ищите в главах 4 и 6 первого тома):

```
# Вызовы методов изменяемого объекта

>>> C
bytearray(b'xYam')

>>> C.append(b'LMN')             # Python 2.X требует строки размером 1
TypeError: an integer is required
Ошибка типа: требуется целое число

>>> C.append(ord('L'))
>>> C
bytearray(b'xYamL')

>>> C.extend(b'MNO')
>>> C
bytearray(b'xYamLMNO')
```

Как и можно было ожидать, с объектами bytearray работают все обычные операции над последовательностями и строковые методы (обратите внимание, что подобно объектам bytes их операции выражений и методы требуют аргументов bytes, не str):

```
# Операции над последовательностями и строковые методы

>>> C
bytearray(b'xYamLMNO')

>>> C + b'!#'
bytearray(b'xYamLMNO!#')
>>> C[0]
120
>>> C[1:]
```

```
bytearray(b'YamLMNO')
>>> len(C)
8

>>> C.replace('xY', 'sp')                          # Работает в Python 2.X
TypeError: Type str doesn't support the buffer API
Ошибка типа: тип str не поддерживает API-интерфейс буфера
>>> C.replace(b'xY', b'sp')
bytearray(b'spamLMNO')

>>> C
bytearray(b'xYamLMNO')
>>> C * 4
bytearray(b'xYamLMNOxYamLMNOxYamLMNOxYamLMNO')
```

Сводка по строковым типам Python 3.X

В заключение для сводки в следующих примерах демонстрируется тот факт, что объекты bytes и bytearray являются последовательностями целых чисел, а объекты str — последовательностями символов:

```
# Двоичные или текстовые

>>> B                  # B - то же, что и S в Python 2.6/2.7
b'spam'
>>> list(B)
[115, 112, 97, 109]

>>> C
bytearray(b'xYamLMNO')
>>> list(C)
[120, 89, 97, 109, 76, 77, 78, 79]

>>> S
'spam'
>>> list(S)
['s', 'p', 'a', 'm']
```

Несмотря на то что все три строковых типа Python 3.X могут содержать значения символов и поддерживают многие те же операции, вы всегда должны:

- использовать тип str для текстовых данных;
- применять тип bytes для двоичных данных;
- использовать тип bytearray для двоичных данных, которые желательно изменять на месте.

Связанные инструменты, такие как файлы, рассматриваемые в следующем разделе, часто делают выбор за вас.

Использование текстовых и двоичных файлов

В этом разделе раскрывается влияние строковой модели Python 3.X на основы обработки файлов, описанные ранее в книге. Как там упоминалось, режим открытия файла критически важен — он определяет, какой тип объекта будет применяться для представления содержимого файла в сценарии. Текстовый режим подразумевает объекты str, а двоичный — объекты bytes.

- Файлы в текстовом режиме интерпретируют файловое содержимое в соответствии с кодировкой Unicode — либо стандартной для вашей платформы, либо той, чье имя вы передали. За счет передачи имени кодировки вызову open вы можете обеспечить принудительные преобразования для различных видов файлов Unicode. Файлы в текстовом режиме также выполняют универсальную трансляцию символов конца строки: по умолчанию все формы символы конца строки отображаются на одиночный символ '\n' в сценарии безотносительно к платформе, на которой сценарий запускается. Как обсуждалось ранее, текстовые файлы поддерживают чтение и запись маркера порядка следования байтов (BOM), хранящегося в начале файла при некоторых схемах кодирования Unicode.
- Файлы в двоичном режиме взамен возвращают файловое содержимое в низкоуровневом формате, как последовательность целых чисел, представляющих значения байтов, без какого-либо кодирования или декодирования и без трансляции символов конца строки.

Второй аргумент вызова open определяет желаемую обработку — текстовую или двоичную, как и в Python 2.X; добавление b к этой строке предполагает двоичный режим (например, "rb" для чтения файлов двоичных данных). Стандартный режим "rt" является тем же, что и "r", который означает текстовый ввод (как принято в Python 2.X).

Тем не менее, в Python 3.X аргумент режима, передаваемый open, также подразумевает *объектный тип* для представления файлового содержимого независимо от лежащей в основе платформы. Текстовые файлы возвращают объект str для операций чтения и ожидают его для операций записи, а двоичные файлы возвращают объект bytes для операций чтения и ожидают его (или bytearray) для операций записи.

Основы текстовых файлов

Давайте начнем демонстрацию с базового файлового ввода-вывода. До тех пор, пока вы обрабатываете базовые текстовые файлы (скажем, ASCII) и не заботитесь об обходе стандартной для платформы кодировки строк, файлы в Python 3.X выглядят почти как в Python 2.X (по существу, как и строки в целом). Например, в следующем взаимодействии Python 3.X в файл записывается одна строка текста, которая затем читается, как делалось бы в Python 3.X (обратите внимание, что file в Python 3.X больше не является встроенным именем, а потому вполне нормально использовать его для переменной):

```
C:\code> C:\python37\python
# Базовые текстовые файлы (и строки) работают таким же образом, как в Python 2.X
>>> file = open('temp', 'w')
>>> size = file.write('abc\n')  # Возвращается количество записанных символов
>>> file.close()                # Ручное закрытие для сброса выходного буфера
>>> file = open('temp')         # Стандартным режимом является "r" (== "rt"):
                                #   текстовый ввод
>>> text = file.read()
>>> text
'abc\n'
>>> print(text)
abc
```

Текстовый и двоичный режимы в Python 2.X и 3.X

В Python 2.X отсутствует значительное разграничение между текстовыми и двоичными файлами – оба вида файлов принимают и возвращают содержимое как строки `str`. Единственное крупное отличие в том, что текстовые файлы автоматически отображают символы конца строки `\n` на `\r\n` и обратно на платформе Windows, тогда как двоичные файлы – нет (для краткости здесь операции приводятся в одной строке):

```
C:\code> C:\python27\python
>>> open('temp', 'w').write('abd\n')  # Запись в текстовом режиме: добавляет \r
>>> open('temp', 'r').read()     # Чтение в текстовом режиме: отбрасывает \r
'abd\n'
>>> open('temp', 'rb').read()         # Чтение в двоичном режиме: дословное
'abd\r\n'

>>> open('temp', 'wb').write('abc\n')  # Запись в двоичном режиме
>>> open('temp', 'r').read()           # \n не расширяется до \r\n
'abc\n'
>>> open('temp', 'rb').read()
'abc\n'
```

В Python 3.X все несколько сложнее из-за различия между типом `str` для текстовых данных и типом `bytes` для двоичных данных. В целях иллюстрации давайте выполним запись в *текстовый файл* и затем чтение из него в обоих режимах в Python 3.X. Важно отметить, что мы обязаны предоставлять объект `str` для записи, но чтение дает нам объект `str` или `bytes` в зависимости от режима, указанного `open`:

```
C:\code> C:\python37\python
# Запись и чтение текстового файла
>>> open('temp', 'w').write('abc\n')   # Вывод в текстовом режиме,
                                       #   предоставить объект str
4
>>> open('temp', 'r').read()   # Ввод в текстовом режиме, возвращает объект str
'abc\n'
>>> open('temp', 'rb').read()  # Ввод в двоичном режиме, возвращает объект bytes
b'abc\r\n'
```

Обратите внимание на то, как на платформе Windows файлы в текстовом режиме при выводе транслируют символ *конца строки* `\n` в `\r\n`; при вводе в текстовом режиме последовательность `\r\n` транслируется обратно в `\n`, но файлы в двоичном режиме трансляцию не производят. В Python 2.X ситуация точно такая же, и обычно это то, что нужно – для переносимости текстовые файлы отображают маркеры конца строки на `\n` и обратно (они фактически присутствуют в файлах Linux, где никакого отображения не происходит), и такие трансляции никогда не должны случаться для двоичных данных (где байты конца строки несущественны). Хотя при желании вы можете управлять указанным поведением с помощью добавочных аргументов `open` в Python 3.X, стандартный вариант обычно работает хорошо.

Давайте сделаем то же самое снова, но с *двоичным файлом*. Теперь для записи мы предоставляем объект `bytes`, а обратно получаем по-прежнему объект `str` или `bytes` в зависимости от режима ввода:

```
# Запись и чтение двоичного файла
>>> open('temp', 'wb').write(b'abc\n') # Вывод в двоичном режиме,
                                       #   предоставить объект bytes
4
```

```
>>> open('temp', 'r').read()  # Ввод в текстовом режиме, возвращает объект str
'abc\n'
>>> open('temp', 'rb').read() # Ввод в двоичном режиме, возвращает объект bytes
b'abc\n'
```

Обратите внимание, что при выводе в двоичном режиме символ конца строки \n не расширяется до \r\n — опять-таки желательный результат для двоичных данных. Требование к типам и поведение файла одинаково, даже если данные, записываемые в двоичный файл, действительно двоичные по своей природе. Скажем, в следующем взаимодействии "\x00" — двоичный ноль и непечатаемый символ:

```
# Запись и чтение по-настоящему двоичных данных
>>> open('temp', 'wb').write(b'a\x00c')     # Предоставить объект bytes
3
>>> open('temp', 'r').read()                # Получение объекта str
'a\x00c'
>>> open('temp', 'rb').read()               # Получение объекта bytes
b'a\x00c'
```

Файлы в двоичном режиме всегда возвращают содержимое как объект bytes, но для записи принимают либо объект bytes, либо объект bytearray; это вполне естественно с учетом того, что bytearray — по существу просто изменяемый вариант bytes. На самом деле большинство API-интерфейсов в Python 3.X, которые принимают объект bytes, также допускают bytearray:

```
# Объекты bytearray тоже подходят
>>> BA = bytearray(b'\x01\x02\x03')

>>> open('temp', 'wb').write(BA)
3
>>> open('temp', 'r').read()
'\x01\x02\x03'
>>> open('temp', 'rb').read()
b'\x01\x02\x03'
```

Несоответствия типов и содержимого в Python 3.X

Обратите внимание, что когда дело касается файлов, то нарушение разграничения между типами str/bytes в Python не остается безнаказанным. Как иллюстрируется в следующих примерах, попытка записи объекта bytes в текстовый файл или объекта str в двоичный файл приводит к получению ошибок (точный текст сообщений об ошибках может изменяться):

```
# Типы не являются гибкими для файлового содержимого
>>> open('temp', 'w').write('abc\n')    # Текстовый режим создает и требует str
4
>>> open('temp', 'w').write(b'abc\n')
TypeError: must be str, not bytes
Ошибка типа: должен быть тип str, не bytes

>>> open('temp', 'wb').write(b'abc\n') # Двоичный режим создает и требует bytes
4
>>> open('temp', 'wb').write('abc\n')
TypeError: 'str' does not support the buffer interface
Ошибка типа: str не поддерживает интерфейс буфера
```

Это имеет смысл: в двоичных терминах текст ничего не значит до того, как он закодирован. Несмотря на то что часто можно выполнять преобразования между ти-

пами, кодируя str и декодируя bytes, как описано ранее в главе, вы обычно будете придерживаться *либо* str для текстовых данных, *либо* bytes для двоичных данных. Поскольку наборы операций типов str и bytes во многом пересекаются, в большинстве программ выбор не станет большой дилеммой (примеры приводятся в финальном разделе главы при рассмотрении инструментов для работы со строками).

Вдобавок к ограничениям типов в Python 3.X может иметь значение и *файловое содержимое*. Выходные файлы в текстовом режиме требуют для своего содержимого объекта str вместо bytes, так что в Python 3.X отсутствует какой-либо способ записи настоящих двоичных данных в файл, открытый в текстовом режиме. В зависимости от правил кодирования байты за пределами стандартного набора символов иногда можно внедрять в нормальную строку, а записывать их в двоичном режиме разрешено всегда (некоторые из показанных ниже примеров генерируют ошибки при отображении их строковых результатов в версиях, предшествующих Python 3.3, но файловые операции работают успешно):

```
# В текстовом режиме нельзя прочитать настоящие двоичные данные
>>> chr(0xFF)                       # FF - допустимый символ, FE - нет
'ÿ'
>>> chr(0xFE)                       # Ошибка в некоторых версиях Python
'\xfe'

>>> open('temp', 'w').write(b'\xFF\xFE\xFD')  # Использовать произвольные
                                              #  байты нельзя!
TypeError: must be str, not bytes
Ошибка типа: должен быть тип str, не bytes

>>> open('temp', 'w').write('\xFF\xFE\xFD')  # Можно записывать, если
                                              #   допускает встраивание в str
3
>>> open('temp', 'wb').write(b'\xFF\xFE\xFD') # Можно также записывать
                                              #   в двоичном режиме
3

>>> open('temp', 'rb').read()       # Всегда можно читать как двоичные байты
b'\xff\xfe\xfd'

>>> open('temp', 'r').read()        # Нельзя читать текст, если он
                                    #  не поддерживает декодирование!
'ÿ\xfe\xfd'                         # Ошибка в некоторых версиях Python
```

Однако в целом из-за того, что в Python 3.X входные файлы в текстовом режиме должны обладать возможностью декодирования содержимого согласно кодировке Unicode, не существует какого-либо способа чтения настоящих двоичных данных в текстовом режиме, как объясняется в следующем разделе.

Использование файлов Unicode

До сих пор мы производили чтение и запись в базовые текстовые и двоичные файлы. Оказывается, читать и записывать текст Unicode, сохраненный в файлах, тоже легко, т.к. вызов open в Python 3.X принимает кодировку для текстовых файлов и организует автоматическое выполнение требуемого кодирования и декодирования при передаче данных. Это позволяет нам обрабатывать разнообразный текст Unicode, который создан посредством различных кодировок, стандартных для платформы, и сохранять тот же самый текст в отличающихся кодировках для разных целей.

Чтение и запись данных Unicode в Python 3.X

На самом деле мы можем эффективно *преобразовывать* строку в другие закодированные формы вручную с помощью вызовов методов, как поступали ранее, и автоматически при файловом вводе и выводе. Для демонстрации в настоящем разделе мы будем применять следующую строку Unicode:

```
C:\code> C:\python37\python
>>> S = 'A\xc4B\xe8C'        # Декодированная строка из пяти символов не ASCII
>>> S
'AÄBèC'
>>> len(S)
5
```

Ручное кодирование

Как уже известно, мы всегда можем закодировать такую строку в низкоуровневые байты в соответствии с именем целевой кодировки:

```
# Кодирование вручную посредством методов
>>> L = S.encode('latin-1')    # 5 байтов при кодировании как latin-1
>>> L
b'A\xc4B\xe8C'
>>> len(L)
5

>>> U = S.encode('utf-8')      # 7 байтов при кодировании как utf-8
>>> U
b'A\xc3\x84B\xc3\xa8C'
>>> len(U)
7
```

Кодирование при файловом выводе

Теперь для записи нашей строки в текстовый файл с заданной кодировкой мы можем просто передать вызову open имя желательной кодировки open — хотя можно было бы сначала закодировать строку вручную и записать ее в двоичном режиме, в этом нет необходимости:

```
# Автоматическое кодирование при записи
>>> open('latindata', 'w', encoding='latin-1').write(S)  # Запись как latin-1
5
>>> open('utf8data', 'w', encoding='utf-8').write(S)     # Запись как utf-8
5

>>> open('latindata', 'rb').read()      # Чтение низкоуровневых байтов
b'A\xc4B\xe8C'

>>> open('utf8data', 'rb').read()       # Строка в файлах отличается
b'A\xc3\x84B\xc3\xa8C'
```

Декодирование при файловом вводе

Аналогично, чтобы прочитать произвольные данные Unicode, мы просто передаем вызову open имя кодировки файла, после чего декодирование низкоуровневых байтов в строки будет производиться автоматически. Мы могли бы также читать низкоуровневые байты и декодировать их вручную, но в случае чтения блоками могут возникнуть сложности (есть вероятность прочитать неполный символ), да и в этом нет нужды:

```
# Автоматическое декодирование при чтении
>>> open('latindata', 'r', encoding='latin-1').read()  # Декодируется при вводе
'AÄBèC'
>>> open('utf8data', 'r', encoding='utf-8').read()    # В соответствии с
                                                      #  указанной кодировкой
'AÄBèC'
>>> X = open('latindata', 'rb').read()          # Ручное декодирование:
>>> X.decode('latin-1')                          # Необязательно
'AÄBèC'
>>> X = open('utf8data', 'rb').read()
>>> X.decode()                                   # UTF-8 - стандартная кодировка
'AÄBèC'
```

Несоответствия при декодировании

Наконец, имейте в виду, что такое поведение файлов в Python 3.X ограничивает тип содержимого, которое можно загружать как текст. В предыдущем разделе было указано, что Python 3.X должен уметь декодировать данные в текстовых файлах в строку str в соответствии со стандартной или переданной кодировкой Unicode. Попытка открыть файл с действительно двоичными данными в текстовом режиме, например, вряд ли будет успешной в Python 3.X, даже в случае использования корректных типов объектов:

```
>>> file = open(r'C:\Python37\python.exe', 'r')
>>> text = file.read()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 2: ...
Ошибка декодирования Unicode: кодек charmap не может декодировать байт 0x90
в позиции 2: ...

>>> file = open(r'C:\Python37\python.exe', 'rb')
>>> data = file.read()
>>> data[:20]
b'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\x00'
```

Первый из приведенных примеров может не потерпеть неудачу в Python 2.X (нормальные файлы не декодируют текст), хотя вероятно должен: чтение файла может возвратить искаженные данные в строке из-за автоматической трансляции символов конца строки в текстовом режиме (при чтении в Windows любые байты \r\n будут переводиться в \n). Чтобы трактовать файловое содержимое как текст Unicode в Python 2.X, вместо встроенной функции open необходимо применять специальные инструменты, как вскоре будет показано. Но сначала давайте займемся более взрывоопасной темой.

Обработка маркера BOM в Python 3.X

Как упоминалось ранее в главе, некоторые схемы кодирования сохраняют в начале файлов специальный маркер BOM для указания порядка байтов (какой конец битовой строки является наиболее значащим для ее значения) или объявления типа кодирования. Интерпретатор Python пропускает этот маркер при вводе и записывает его при выводе, если имя кодировки подразумевает его наличие, но временами мы должны использовать специфическое имя кодировки, чтобы явно обеспечить обработку BOM.

Скажем, в кодировках UTF-16 и UTF-32 маркер BOM задает формат с прямым или обратным порядком следования байтов. Текстовый файл UTF-8 также может включать маркер BOM, но его наличие не гарантируется и он служит только для объявления в целом кодировки UTF-8. При чтении и записи данных с применением таких схем ко-

дирования интерпретатор Python автоматически пропускает или записывает маркер BOM, если он подразумевается именем универсальной кодировки или если вы предоставляете имя более специфической кодировки, чтобы решить задачу. Например:

- в UTF-16 маркер BOM всегда обрабатывается для `utf-16`, а имя более специфической кодировки `utf-16-le` обозначает формат с прямым порядком следования байтов;
- в UTF-8 более специфическая кодировка `utf-8-sig` вынуждает интерпретатор Python пропускать и записывать маркер BOM при вводе и выводе, но универсальная кодировка `utf-8` — нет.

Отбрасывание маркера BOM в редакторе "Блокнот"

Давайте создадим несколько файлов с маркерами BOM, чтобы посмотреть, как все работает на практике. Когда вы сохраняете текстовый файл в редакторе "Блокнот" в среде Windows, то можете выбрать из раскрывающегося списка тип его кодировки — простой текст ASCII, UTF-8 либо UTF-16 с прямым или обратным порядком следования байтов. Скажем, если в редакторе "Блокнот" сохраняется двухстрочный текстовый файл по имени `spam.txt` с типом кодировки *ANSI*, тогда он записывается как простой текст ASCII без маркера BOM. При чтении этого файла в Python в двоичном режиме мы можем видеть фактические байты, сохраненные в файле. Когда файл читается в текстовом режиме, интерпретатор Python по умолчанию выполняет трансляцию символов конца строки. Мы также можем его декодировать как явный текст UTF-8, поскольку ASCII представляет собой подмножество данной схемы (или надмножество cp1252 из Latin-1, которое является стандартной кодировкой в Windows для `open` в Python 3.X согласно `locale.getpreferredencoding`):

```
C:\code> C:\python37\python          # Файл, сохраненный в редакторе 'Блокнот'
>>> import sys, locale
>>> locale.getpreferredencoding(False)
'cp1252'
>>> open('spam.txt', 'rb').read()  # Текстовый файл ASCII (UTF-8)
b'spam\r\nSPAM\r\n'
>>> open('spam.txt', 'r').read()   # Текстовый режим транслирует символы
                                   #   конца строки
'spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'spam\nSPAM\n'
```

Если взамен сохранить файл `spam.txt` в редакторе "Блокнот" как *UTF-8*, тогда в его начало добавится 3-байтовая последовательность маркера BOM для UTF-8, а нам придется предоставить имя более специфической кодировки (`utf-8-sig`), чтобы заставить интерпретатор Python пропустить маркер:

```
>>> open('spam.txt', 'rb').read()  # UTF-8 с 3-байтовым маркером BOM
b'\xef\xbb\xbfspam\r\nSPAM\r\n'
>>> open('spam.txt', 'r').read()
'ï»¿spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'\ufeffspam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8-sig').read()
'spam\nSPAM\n'
```

Если сохранить файл в редакторе "Блокнот" как *Unicode с обратным порядком следования байтов*, то мы получим в файле данные в формате UTF-16 с 2-байтовыми (16-бит-

ными) символами, которые предваряются 2-байтовой последовательностью маркера BOM. Имя кодировки utf-16 в Python обеспечит пропуск маркера BOM, т.к. его наличие подразумевается (поскольку все файлы UTF-16 имеют BOM), а имя кодировки utf-16-be позволит обрабатывать формат с обратным порядком следования байтов, но не пропускать BOM (второй из показанных далее примеров потерпит неудачу в более старых версиях Python):

```
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00s\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n'
>>> open('spam.txt', 'r').read()
'\xfeÿ\x00s\x00p\x00a\x00m\x00\n\x00\n\x00S\x00P\x00A\x00M\x00\n\x00\n'
>>> open('spam.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-16-be').read()
'\ufeffspam\nSPAM\n'
```

Кстати, кодировка Unicode в редакторе "Блокнот" представляет собой UTF-16 с прямым порядком следования байтов (одну из очень многих видов кодировки Unicode!).

Отбрасывание маркера BOM в Python

Те же самые схемы в целом остаются справедливыми и для *вывода*. При записи в файл Unicode из кода на Python нам понадобится более явное имя кодировки для обеспечения маркера BOM в UTF-8 — utf-8 не записывает BOM, но utf-8-sig записывает:

```
>>> open('temp.txt', 'w', encoding='utf-8').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()        # Маркер BOM отсутствует
b'spam\r\nSPAM\r\n'

>>> open('temp.txt', 'w', encoding='utf-8-sig').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()        # Записывает маркер BOM
b'\xef\xbb\xbfspam\r\nSPAM\r\n'

>>> open('temp.txt', 'r').read()
'ï»¿spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()    # Сохраняет маркер BOM
'\ufeffspam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read() # Пропускает маркер BOM
'spam\nSPAM\n'
```

Обратите внимание, что хотя utf-8 не отбрасывает BOM, данные *без* маркера BOM могут быть прочитаны посредством utf-8 и utf-8-sig — используйте последнюю кодировку для ввода, если вы не уверены, присутствует ли маркер BOM в файле:

```
>>> open('temp.txt', 'w').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()             # Данные без маркера BOM
b'spam\r\nSPAM\r\n'

>>> open('temp.txt', 'r').read()              # Работает и utf-8, и utf-8-sig
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()
'spam\nSPAM\n'
```

Наконец, для имени кодировки `utf-16` маркер BOM обрабатывается автоматически: при *выводе* данные записываются с естественным для платформы порядком байтов и маркер BOM всегда добавляется; при *вводе* данные декодируются согласно BOM и маркер BOM всегда разбирается, т.к. он стандартный в этой схеме:

```
>>> sys.byteorder
'little'
>>> open('temp.txt', 'w', encoding='utf-16').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()
b'\xff\xfes\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n\x00'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'
```

Более специфические кодировки UTF-16 способны задавать другой порядок байтов, хотя в ряде сценариев вы можете вручную записывать и пропускать маркер BOM, если он обязателен или присутствует — для получения добавочных инструкций относительно создания BOM изучите приведенные ниже примеры:

```
>>> open('temp.txt', 'w', encoding='utf-16-be').write('\ufeffspam\nSPAM\n')
11
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00s\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-16-be').read()
'\ufeffspam\nSPAM\n'
```

Более специфические кодировки UTF-16 хорошо работают с файлами, не содержащими маркера BOM, но `utf-16` требует его наличия при вводе, чтобы выяснить порядок следования байтов:

```
>>> open('temp.txt', 'w', encoding='utf-16-le').write('SPAM')
4
>>> open('temp.txt', 'rb').read()  # Нормально, если BOM не присутствует
                                   #   или не ожидается
b'S\x00P\x00A\x00M\x00'
>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'SPAM'
>>> open('temp.txt', 'r', encoding='utf-16').read()
UnicodeError: UTF-16 stream does not start with BOM
```

Поэкспериментируйте с кодировками самостоятельно или обратитесь в руководство по библиотеке Python за дополнительными сведениями о маркере BOM.

Файлы Unicode в Python 2.X

Предыдущее обсуждение применимо к строковым типам и файлам Python 3.X. Добиться похожего эффекта для файлов Unicode можно и в Python 2.X, но интерфейс будет другим. Тем не менее, если вы замените `str` типом `unicode`, а `open` вызовом `codecs.open`, то результат окажется по существу таким же, как в Python 3.X:

```
C:\code> C:\python27\python
>>> S = u'A\xc4B\xe8C'           # Тип Python 2.X
>>> print S
'AÄBèC'
>>> len(S)
5
```

```
>>> S.encode('latin-1')                          # Вызовы вручную
'A\xc4B\xe8C'
>>> S.encode('utf-8')
'A\xc3\x84B\xc3\xa8C'

>>> import codecs                                # Файлы Python 2.X
>>> codecs.open('latindata', 'w', encoding='latin-1').write(S)
                                    # Записать закодированную строку
>>> codecs.open('utfdata', 'w', encoding='utf-8').write(S)

>>> open('latindata', 'rb').read()
'A\xc4B\xe8C'
>>> open('utfdata', 'rb').read()
'A\xc3\x84B\xc3\xa8C'

>>> codecs.open('latindata', 'r', encoding='latin-1').read()
                                    # Прочитать декодированную строку
u'A\xc4B\xe8C'
>>> codecs.open('utfdata', 'r', encoding='utf-8').read()
u'A\xc4B\xe8C'
>>> print codecs.open('utfdata', 'r', encoding='utf-8').read()
                                    # Вывести для просмотра
'AÄBèC'
```

Дополнительные детали, касающиеся Unicode в Python 2.X, ищите в предшествующих разделах главы и в руководстве по Python 2.X.

Имена файлов и потоки данных Unicode

В заключение отметим, что хотя раздел был посвящен кодированию и декодированию *содержимого* текстовых файлов Unicode, в Python также поддерживается понятие *имен* файлов не ASCII. На самом деле они являются независимыми настройками в sys, которые могут варьироваться в зависимости от версии Python и платформы (интерпретатор Python 2.X возвращает для файлового содержимого кодировку ASCII в Windows):

```
>>> import sys
>>> sys.getdefaultencoding(), sys.getfilesystemencoding()
                                    # Файловое содержимое, имена
('utf-8', 'mbcs')
```

Имена файлов: текст или байты

Кодирование имен файлов часто не является проблемой. Выражаясь кратко, для имен файлов, заданных как строки текста Unicode, вызов open выполняет кодирование в соответствии с соглашениями об именах файлов, принятыми на текущей платформе. Передача файловым инструментам (в числе которых open и средства для прохода по каталогам и построения списков файлов) произвольно закодированных имен файлов в виде байтовых строк переопределяет автоматическое кодирование и вынуждает возвращать имена файлов тоже в форме закодированных байтовых строк. Это полезно, когда имена файлов не допускают декодирование по соглашениям лежащей в основе платформы (я работаю в Windows, но какие-то из следующих примеров могут потерпеть неудачу на других платформах):

```
>>> f = open('xxx\u00A5', 'w')          # Имя файла не ASCII
>>> f.write('\xA5999\n')                # Записать пять символов
>>> f.close()
>>> print(open('xxx\u00A5').read())     # Текст: кодируется автоматически
¥999
```

```
>>> print(open(b'xxx\xA5').read())        # Байты: кодируется заранее
¥999

>>> import glob                           # Инструмент расширения имен файлов
>>> glob.glob('*\u00A5*')                 # Получает закодированный текст
                                          #  для закодированного текста
['xxx¥']
>>> glob.glob(b'*\xA5*')                  # Получает закодированные байты
                                          #  для закодированных байтов
[b'xxx\xa5']
```

Содержимое потоков данных: PYTHONIOENCODING

Кроме того, с помощью переменной среды PYTHONIOENCODING можно устанавливать кодировку, используемую для текста в стандартных *потоках данных* – ввода, вывода и ошибок. Эта настройка переопределяет стандартную кодировку Python для отображаемого текста, которая на платформе Windows в настоящее время применяет формат Windows в Python 3.X и ASCII в Python 2.X. Установка переменной среды PYTHONIOENCODING в универсальный формат Unicode вроде UTF-8 иногда может требоваться для вывода текста, отличающегося от ASCII, и для отображения такого текста в окнах командной оболочки (возможно в сочетании с изменениями кодовой страницы на некоторых компьютерах Windows). Например, если подобная установка не сделана, тогда сценарий, выводящий имена файлов не ASCII, может потерпеть неудачу.

Дополнительные сведения по данной теме можно найти в разделе "Символы валют: Unicode в действии" главы 25 первого тома. Там прорабатывался пример, демонстрирующий сущность переносимого кодирования Unicode, а также роли и требования к настройке PYTHONIOENCODING, к которым мы здесь возвращаться не будем.

В целом такие темы раскрываются в руководствах по Python или книгах вроде *Programming Python, 4th Edition* (http://www.oreilly.com/catalog/9780596158101), где более глубоко исследуются потоки данных и файлы с прикладной точки зрения.

Другие изменения инструментов для обработки строк в Python 3.X

Многие другие популярные инструменты обработки строк в стандартной библиотеке Python также были модернизированы с целью учета нового противопоставления между типами str и bytes. Мы не собирались особо подробно раскрывать эти ориентированные на прикладные приложения инструменты в книге, посвященной базовому языку, но в завершение главы кратко рассмотрим четыре крупных инструмента, которые были подвергнуты воздействию: модуль re для сопоставления с образцом, модуль struct для работы с двоичными данными, модуль pickle для сериализации объектов и пакет xml для разбора текста XML. Как будет отмечаться далее, другие инструменты Python, такие как модуль json, отличаются в аспектах, подобных тем, что представлены здесь.

Модуль re для сопоставления с образцом

Модуль re для сопоставления с образцом в Python поддерживает более универсальную обработку текста, нежели та, которую обеспечивают вызовы простых строковых методов наподобие find, split и replace. С помощью модуля re строки, обозначающие цели поиска и разделения, могут быть описаны универсальными образцами, а не абсолютным текстом. Модуль re был обобщен для работы с объектами любого стро-

кового типа в Python 3.X — str, bytes и bytearray — и возвращает результирующие подстроки того же самого типа, что и исходная строка. В Python 2.X он поддерживает типы unicode и str.

Ниже иллюстрируется работа модуля re в Python 3.X при извлечении подстрок из строки текста, взятого естественно из кинофильма "Monty Python's The Meaning of Life" ("Смысл жизни по Монти Пайтону"). Внутри строк образцов (.*) означает любой символ (.) ноль или больше раз (*), сохраненный как совпавшая подстрока (()). Части строки, которые дали совпадение по частям образца, заключенным в круглые скобки, доступны после успешного сопоставления через метод group или groups:

```
C:\code> C:\python37\python
>>> import re
>>> S = 'Bugger all down here on earth!'          # Строка текста
>>> B = b'Bugger all down here on earth!'         # Обычно поступает из файла

>>> re.match('(.*) down (.*) on (.*)', S).groups()  # Сопоставление строки
                                                     #   с образцом
('Bugger all', 'here', 'earth!')                  # Совпавшие подстроки

>>> re.match(b'(.*) down (.*) on (.*)', B).groups() # Подстроки типа bytes
(b'Bugger all', b'here', b'earth!')
```

В Python 2.X результаты аналогичны, но для отличающегося от ASCII текста используется тип unicode, а тип str поддерживает 8-битный и двоичный текст:

```
C:\code> C:\python27\python
>>> import re
>>> S = 'Bugger all down here on earth!'      # Простой и двоичный текст
>>> U = u'Bugger all down here on earth!'     # Текст Unicode

>>> re.match('(.*) down (.*) on (.*)', S).groups()
('Bugger all', 'here', 'earth!')

>>> re.match('(.*) down (.*) on (.*)', U).groups()
(u'Bugger all', u'here', u'earth!')
```

Поскольку типы bytes и str поддерживают по существу одинаковые наборы операций, такое различение между типами почти совершенно прозрачно. Но обратите внимание, как и в других API-интерфейсах, вы не можете смешивать типы str и bytes в аргументах их вызовов в Python 3.X (хотя если вы не планируете выполнять сопоставлении с образцом на двоичных данных, то вероятно вам и не нужно беспокоиться):

```
C:\code> C:\python37\python
>>> import re
>>> S = 'Bugger all down here on earth!'
>>> B = b'Bugger all down here on earth!'

>>> re.match('(.*) down (.*) on (.*)', B).groups()
TypeError: can't use a string pattern on a bytes-like object
Ошибка типа: нельзя использовать строковый образец с объектом, подобным bytes

>>> re.match(b'(.*) down (.*) on (.*)', S).groups()
TypeError: can't use a bytes pattern on a string-like object
Ошибка типа: нельзя использовать образец bytes с объектом, подобным строке

>>> re.match(b'(.*) down (.*) on (.*)', bytearray(B)).groups()
(bytearray(b'Bugger all'), bytearray(b'here'), bytearray(b'earth!'))

>>> re.match('(.*) down (.*) on (.*)', bytearray(B)).groups()
TypeError: can't use a string pattern on a bytes-like object
Ошибка типа: нельзя использовать строковый образец с объектом, подобным bytes
```

Модуль struct для работы с двоичными данными

Модуль struct, применяемый для создания и извлечения упакованных двоичных данных из строк, работает в Python 3.X так же, как в Python 2.X, но упакованные данные в Python 3.X представляются только в виде объектов bytes и bytearray, но не str (что имеет смысл, учитывая его ориентацию на обработку двоичных данных, а не декодированного текста); значения кодов данных s должны быть bytes, начиная с версии Python 3.2 (предшествующее автоматическое кодирование UTF-8 в str отброшено).

Ниже демонстрируется упаковка трех объектов в строку в обеих линейках Python согласно спецификации двоичных типов (здесь создаются 4-байтовое целое число, 4-байтовая строка и 2-байтовое целое число):

```
C:\code> C:\python37\python
>>> from struct import pack
>>> pack('>i4sh', 7, b'spam', 8)     # bytes в Python 3.X (8-битные строки)
b'\x00\x00\x00\x07spam\x00\x08'

C:\code> C:\python27\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'spam', 8)      # str в Python 2.X (8-битные строки)
'\x00\x00\x00\x07spam\x00\x08'
```

Однако так как bytes имеет интерфейс, почти идентичный интерфейсу str в Python 3.X и 2.X, большинству программистов видимо не придется беспокоиться — изменение не касается большей части существующего кода, особенно из-за того, что операция чтения двоичного файла автоматически создает объект bytes. Хотя последний тест в следующем примере терпит неудачу по причине несовпадения типов, большинство сценариев будут читать двоичные данные из файла, не создавая их в виде строки, как мы поступаем здесь:

```
C:\code> C:\python37\python
>>> import struct
>>> B = struct.pack('>i4sh', 7, b'spam', 8)
>>> B
b'\x00\x00\x00\x07spam\x00\x08'

>>> vals = struct.unpack('>i4sh', B)
>>> vals
(7, b'spam', 8)

>>> vals = struct.unpack('>i4sh', B.decode())
TypeError: 'str' does not support the buffer interface
Ошибка типа: str не поддерживает интерфейс буфера
```

Помимо нового синтаксиса для байтов создание и чтение двоичных файлов работает в Python 3.X почти так же, как в Python 2.X. Тем не менее, код вроде показанного далее является одним из главных мест, где программисты заметят объектный тип bytes:

```
C:\code> C:\python37\python
# Запись значений в упакованный двоичный файл
>>> F = open('data.bin', 'wb')                # Открыть выходной двоичный файл
>>> import struct
>>> data = struct.pack('>i4sh', 7, b'spam', 8)   # Создать упакованные
                                                 #   двоичные данные
>>> data                                      # bytes в Python 3.X, не str
b'\x00\x00\x00\x07spam\x00\x08'
```

```
>>> F.write(data)                              # Записать в файл
10
>>> F.close()

# Чтение значений из упакованного двоичного файла
>>> F = open('data.bin', 'rb')                 # Открыть входной двоичный файл
>>> data = F.read()                            # Читать байты
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> values = struct.unpack('>i4sh', data)  # Извлечь упакованные двоичные данные
>>> values                                     # Вернуться к объектам Python
(7, b'spam', 8)
```

После извлечения упакованных данных в объекты Python вы можете при желании еще глубже погрузиться в двоичный мир – строки разрешено индексировать и нарезать для получения значений индивидуальных байтов, с помощью побитовых операций из целых чисел можно извлекать отдельные биты и т.д. (применяемые здесь операции были описаны ранее в книге):

```
>>> values                      # Результат struct.unpack
(7, b'spam', 8)

# Доступ к битам разобранных целых чисел
>>> bin(values[0])              # Можно добраться до битов в целых числах
'0b111'
>>> values[0] & 0x01            # Проверить первый (самый младший) бит в
целом числе
1
>>> values[0] | 0b1010          # Побитовое ИЛИ: включить биты
15
>>> bin(values[0] | 0b1010)  # Десятичное число 15 соответствует двоичному 1111
'0b1111'
>>> bin(values[0] ^ 0b1010)     # Побитовое исключающее ИЛИ: отключить,
                                #   если оба истинны
'0b1101'
>>> bool(values[0] & 0b100)     # Проверить, включен ли бит 3
True
>>> bool(values[0] & 0b1000)    # Проверить, включен ли бит 4
False
```

Поскольку разобранные строки `bytes` представляют собой последовательности коротких целых чисел, мы можем организовать похожую обработку для их индивидуальных байтов:

```
# Доступ к байтам разобранных строк и битам внутри них
>>> values[1]
b'spam'
>>> values[1][0]             # Строка bytes: последовательность целых чисел
115
>>> values[1][1:]            # Выводится как символы ASCII
b'pam'
>>> bin(values[1][0])        # Можно добраться до битов внутри байтов в
строках
'0b1110011'
>>> bin(values[1][0] | 0b1100)    # Включить биты
'0b1111111'
>>> values[1][0] | 0b1100
127
```

Конечно, большинство программистов на Python не имеют дело с двоичными битами; в Python имеются объектные типы более высокого уровня, подобные спискам и словарям, которые обычно будут лучшим выбором для представления информации в сценариях Python. Однако если вы обязаны использовать или производить низкоуровневые данные, потребляемые программами на С, библиотеками работы с сетями или другими интерфейсами, то Python располагает инструментами, которые помогут в этой.

Модуль pickle для сериализации объектов

Мы бегло сталкивались с модулем pickle в главе 9 первого тома и дополнительно в главах 28 и 31. В главе 28 мы также применяли модуль shelve, который внутренне использует pickle. Для полноты картины имейте в виду, что версия модуля pickle из Python 3.X всегда создает объект bytes независимо от стандартного или переданного "протокола" (уровень формата данных). Вы можете увидеть его с применением вызова dumps модуля pickle, который возвращает строку с сериализированным объектом:

```
C:\code> C:\python37\python
>>> import pickle                  # dumps() возвращает строку
                                   #   с сериализированным объектом
>>> pickle.dumps([1, 2, 3])        # Стандартный протокол Python 3.X: 3=двоичный
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'

>>> pickle.dumps([1, 2, 3], protocol=0) # Протокол ASCII 0, но по-прежнему bytes!
b'(lp0\nL1L\naL2L\naL3L\na.'
```

В итоге подразумевается, что файлы, используемые для хранения обработанных модулем pickle объектов, в Python 3.X всегда должны открываться в *двоичном режиме*, т.к. текстовые файлы для представления данных применяют строки str, не bytes — вызов dump просто пытается записать строку с сериализированными данными в открытый выходной файл:

```
>>> pickle.dump([1, 2, 3], open('temp', 'w')) # Текстовые файлы терпят неудачу
                                                в случае bytes!
TypeError: must be str, not bytes             # Вопреки значению протокола
Ошибка типа: должен быть тип str, не bytes

>>> pickle.dump([1, 2, 3], open('temp', 'w'), protocol=0)
TypeError: must be str, not bytes
Ошибка типа: должен быть тип str, не bytes

>>> pickle.dump([1, 2, 3], open('temp', 'wb')) # В Python 3.X всегда используется
                                              #   двоичный режим

>>> open('temp', 'r').read()          # Работает, но по счастливой случайности
'\u20ac\x03]q\x00(K\x01K\x02K\x03e.'
```

Обратите внимание, что последний результат здесь не заканчивается ошибкой в текстовом режиме лишь потому, что хранящиеся двоичные данные были совместимыми со стандартной кодировкой UTF-8 платформы Windows; вообще говоря, нам просто повезло (в действительности эта команда потерпит неудачу при выводе в более старых версиях Python и может отказать на других платформах). Поскольку данные, выдаваемые pickle, обычно не являются декодируемым текстом Unicode, то же самое правило остается справедливым при вводе — корректное использование в Python 3.X требует записывания и чтения данных pickle в двоичном режиме независимо от того, будет предприниматься последующая десериализация или нет:

```
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))
>>> pickle.load(open('temp', 'rb'))
[1, 2, 3]
>>> open('temp', 'rb').read()
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

В Python 2.X для данных pickle мы можем обойтись файлами в текстовом режиме при условии применения протокола уровня 0 (стандарт в Python 2.X) и согласованного использования текстового режима для преобразования символов конца строки:

```
C:\code> C:\python27\python
>>> import pickle
>>> pickle.dumps([1, 2, 3])        # Стандартный протокол Python 2.X: 0=ASCII
'(lp0\nI1\naI2\naI3\na.'

>>> pickle.dumps([1, 2, 3], protocol=1)
']q\x00(K\x01K\x02K\x03e.'

>>> pickle.dump([1, 2, 3], open('temp', 'w'))    # Текстовый режим в Python
                                                 #   2.X работает
>>> pickle.load(open('temp'))
[1, 2, 3]
>>> open('temp').read()
'(lp0\nI1\naI2\naI3\na.'
```

Тем не менее, если вас интересует нейтральность к версии, либо вы не хотите заботиться о протоколах или их стандартных значениях, специфичных к версии, тогда для данных, обработанных модулем pickle, всегда применяйте файлы в двоичном режиме. Следующий прием работает одинаково в Python 3.X и 2.X:

```
>>> import pickle
>>> pickle.dump([1, 2, 3], open('temp', 'wb')) # Подход, нейтральный к версии
>>> pickle.load(open('temp', 'rb'))            # И обязательный в Python 3.X
[1, 2, 3]
```

Из-за того, что почти все программы позволяют интерпретатору Python сериализировать и десериализировать объекты посредством модуля pickle автоматически и не имеют дела с содержимым самих данных pickle, требование всегда использовать двоичный режим для файлов представляет собой единственную значительную несовместимость в более новой модели сериализации Python 3.X. Дополнительные сведения о сериализации объектов ищите в справочниках или в руководстве по Python.

Инструменты для разбора XML

XML – это основанный на дескрипторах язык для описания структурированной информации, широко применяемый для определения документов и данных, которые передаются через веб-сеть. Несмотря на то что некоторую информацию можно извлечь из текста XML с помощью базовых строковых методов или модуля re, вложение конструкций и текста произвольных атрибутов XML требуют более высокой точности от полного разбора.

Поскольку XML является настолько распространенным форматом, сам Python поступает с целым пакетом инструментов для разбора XML, которые поддерживают модели разбора SAX и DOM, а также пакет, известный как *ElementTree* – API-интерфейс Python, предназначенный для разбора и построения XML. Помимо базового разбора сообщество открытого кода предлагает дополнительные инструменты для работы с XML, такие как XPath, XQuery, XSLT и т.д.

По определению XML представляет текст в форме Unicode, чтобы поддерживать интернационализацию. Хотя большинство инструментов разбора XML языка Python всегда возвращают строки Unicode, в Python 3.X их результаты видоизменились из типа `unicode` версии Python 2.X в универсальный строковый тип `str` версии Python 3.X. Это имеет смысл, принимая во внимание тот факт, что строка `str` в Python 3.X *представлена* в виде Unicode, будь кодировкой ASCII или какая-то другая.

Здесь мы не будем вдаваться в особые детали, но чтобы продемонстрировать суть, давайте возьмем простой текстовый файл XML по имени `mybooks.xml`:

```
<books>
   <date>1995~2013</date>
   <title>Learning Python</title>
   <title>Programming Python</title>
   <title>Python Pocket Reference</title>
   <publisher>O'Reilly Media</publisher>
</books>
```

Мы хотим запустить сценарий для извлечения и отображения содержимого всех вложенных дескрипторов `title` следующим образом:

```
Learning Python
Programming Python
Python Pocket Reference
```

Существуют, по меньшей мере, четыре базовых способа решения задачи (не считая более сложных инструментов вроде XPath). Во-первых, мы могли бы выполнить *сопоставление с образцом* для текста в файле, но оно может оказаться неточным при непрогнозируемом тексте. В случае применимости обсуждавшийся ранее модуль `re` справится с работой — его метод `match` ищет совпадение в начале строки, метод `search` просматривает вперед в поисках совпадения, а используемый ниже метод `findall` находит все места в строке, где образец дает совпадение (результат возвращается в виде списка совпадающих подстрок, соответствующих заключенным в круглые скобки группам образца, либо кортежей такого рода для множества групп):

```
# Файл patternparse.py

import re
text = open('mybooks.xml').read()
found = re.findall('<title>(.*)</title>', text)
for title in found: print(title)
```

Во-вторых, для большей надежности мы могли бы выполнить полный разбор XML с помощью поддержки *разбора DOM* из стандартной библиотеки. Разбор DOM превращает текст XML в дерево объектов и предоставляет интерфейс для навигации по дереву с целью извлечения атрибутов и значений дескрипторов; этот интерфейс является формальной спецификацией, независимой от Python:

```
# Файл domparse.py

from xml.dom.minidom import parse, Node
   xmltree = parse('mybooks.xml')
   for node1 in xmltree.getElementsByTagName('title'):
      for node2 in node1.childNodes:
         if node2.nodeType == Node.TEXT_NODE:
            print(node2.data)
```

В-третьих, стандартная библиотека Python поддерживает *разбор SAX* для XML. В рамках модели SAX методы класса получают обратные вызовы по мере продвиже-

ния разбора и используют информацию состояния для отслеживания места, где они находятся в документе, и накопления его данных:

```
# Файл saxparse.py

import xml.sax.handler
class BookHandler(xml.sax.handler.ContentHandler):
   def __init__(self):
      self.inTitle = False
   def startElement(self, name, attributes):
      if name == 'title':
         self.inTitle = True
   def characters(self, data):
      if self.inTitle:
         print(data)
   def endElement(self, name):
      if name == 'title':
         self.inTitle = False

import xml.sax
parser = xml.sax.make_parser()
handler = BookHandler()
parser.setContentHandler(handler)
parser.parse('mybooks.xml')
```

Наконец, в-четвертых, система *ElementTree*, доступная в пакете `etree` стандартной библиотеки, часто способна обеспечить такие же результаты, как инструменты разбора XML DOM, но со значительно меньшим объемом кода. Она является специфичным для Python способом разбора и генерации текста XML; после разбора ее API-интерфейс предоставляет доступ к компонентам документа:

```
# Файл etreeparse.py

from xml.etree.ElementTree import parse
tree = parse('mybooks.xml')
for E in tree.findall('title'):
   print(E.text)
```

В случае запуска под управлением Python 2.X или 3.X все четыре сценария отображают одинаковые результаты:

```
C:\code> C:\python27\python domparse.py
Learning Python
Programming Python
Python Pocket Reference

C:\code> C:\python37\python domparse.py
Learning Python
Programming Python
Python Pocket Reference
```

Однако формально в Python 2.X некоторые сценарии создают строковые объекты `unicode`, тогда как в Python 3.X все они производят строки `str`, поскольку тип `str` включает текст Unicode (будь он ASCII или другим):

```
C:\code> C:\python37\python
>>> from xml.dom.minidom import parse, Node
>>> xmltree = parse('mybooks.xml')
>>> for node in xmltree.getElementsByTagName('title'):
      for node2 in node.childNodes:
```

```
        if node2.nodeType == Node.TEXT_NODE:
            node2.data

'Learning Python'
'Programming Python'
'Python Pocket Reference'

C:\code> C:\python27\python
>>> ...тот же самый код...

u'Learning Python'
u'Programming Python'
u'Python Pocket Reference'
```

Программы, которые обязаны обрабатывать результаты разбора XML нетривиальными способами, должны будут учитывать отличающийся объектный тип в Python 3.X. Тем не менее, из-за того, что все строки имеют приблизительно идентичные интерфейсы в Python 2.X и 3.X, изменение не затронет большинство сценариев; инструменты, доступные для `unicode` в Python 2.X, как правило, доступны для `str` в Python 3.X. Основной трюк, если уж на то пошло, связан с получением имен кодировок прямо при передаче разобранных данных в файлы и из файлов, сетевые подключения, графические пользовательские интерфейсы и т.п.

К сожалению, рассмотрение дальнейших деталей разбора XML выходит за рамки настоящей книги. Если вы заинтересованы в изучении разбора текста или XML, тогда обратитесь к книге, ориентированной на прикладные приложения, такой как *Programming Python* (`http://www.oreilly.com/catalog/9780596158101`). Дополнительные сведения о `re`, `struct`, `pickle` и XML, а также о влиянии Unicode на остальные библиотечные инструменты вроде средств для расширения имен файлов и просмотра каталогов ищите в веб-сети, в упомянутой выше книге и в руководстве по стандартной библиотеке Python.

В качестве связанной темы проработайте также пример применения *JSON* в главе 9 первого тома — нейтрального к языку формата обмена данными. Его структура очень похожа на словари и списки Python, а все строки JSON представлены в формате Unicode, который отличается по типу между Pythons 2.X и 3.X во многом так же, как было показано здесь для XML.

Что потребует внимания: инспектирование файлов и многое другое

При обновлении этой главы я был сбит с толку одним сценарием использования для ряда рассматриваемых в ней инструментов. После сохранения HTML-файла, ранее бывшего ASCII, в редакторе "Блокнот" как UTF8 я обнаружил, что в процессе он обзавелся мистическим символом не ASCII из-за явной ошибки ввода с клавиатуры и больше не воспринимается как ASCII текстовыми инструментами. Чтобы найти недействительный символ, я просто запустил интерпретатор Python, декодировал содержимое файла из формата UTF-8, открыв его в *текстовом режиме*, и просмотрел символ за символом в поисках первого байта, который не был допустимым символом ASCII:

```
>>> f = open('py33-windows-launcher.html', encoding='utf8')
>>> t = f.read()
>>> for (i, c) in enumerate(t):
        try:
            x = c.encode(encoding='ascii')
        except:
            print(i, sys.exc_info()[0])
    9886 <class 'UnicodeEncodeError'>
```

Располагая индексом недействительного символа, легко получить срез строки Unicode для выяснения дополнительных деталей:

```
>>> len(t)
31021
>>> t[9880:9890]
'ugh. \u206cThi'
>>> t[9870:9890]
'trace through. \u206cThi'
```

После исправления я мог бы также открыть файл в *двоичном режиме*, чтобы продолжить проверку и исследование фактического незакодированного содержимого файла:

```
>>> f = open('py33-windows-launcher.html', 'rb')
>>> b = f.read()
>>> b[0]
60
>>> b[:10]
b'<HTML>\r\n<T'
```

Возможно, это не та работа, которая требует незаурядных умственных способностей, и существуют другие подходы, но Python в таких случаях предоставляет удобный тактический инструмент, а его файловые объекты при необходимости дают вам осязаемое окно в данные, как в сценариях, так и в интерактивном режиме.

Хотя Unicode несомненно является важной темой в нынешнем мире глобального программного обеспечения, он более обязателен, чем вы могли ожидать, особенно в языке вроде Python 3.X, который поднимает его до своих основных строковых и файловых типов, в итоге приводя всех пользователей в сообщество Unicode — готовы они к этому или нет!

Резюме

В главе детально исследовались расширенные строковые типы, доступные в Python 3.X и 2.X для обработки текста Unicode и двоичных данных. Как выяснилось, многие программисты применяют текст ASCII и могут обойтись базовым строковым типом и его операциями. Для более сложных приложений строковая модель Python полностью поддерживает обогащенный текст Unicode (через нормальный строковый тип в Python 3.X и специальный тип в Python 2.X) и байтовые данные (представляемые с помощью типа bytes в Python 3.X и нормальных строк в Python 2.X).

Кроме того, было показано, как файловый объект Python видоизменился в Python 3.X, чтобы автоматически кодировать и декодировать текст Unicode и иметь дело с байтовыми строками для файлов в двоичном режиме, и описаны похожие средства для Python 2.X. Наконец, было кратко представлено несколько инструментов для работы с текстовыми и двоичными данными в библиотеке Python, а также приведены примеры их использования в Python 3.X и 2.X.

В следующей главе мы сосредоточим внимание на темах, связанных с построением инструментов, и рассмотрим способы управления доступом к атрибутам объектов путем вставки автоматически запускаемого кода. Но прежде чем двигаться дальше, закрепите пройденный материал главы, ответив на контрольные вопросы. Поскольку это была важная глава, обязательно ознакомьтесь с ответами на вопросы, чтобы получить более полное представление о том, что обсуждалось в главе.

Проверьте свои знания: контрольные вопросы

1. Каковы имена и роли типов строковых объектов в Python 3.X?
2. Каковы имена и роли типов строковых объектов в Python 2.X?
3. Как соответствуют друг другу строковые типы в Python 2.X и Python 3.X?
4. Чем отличаются строковые типы Python 3.X с точки зрения операций?
5. Как можно записать символы Unicode, отличающиеся от ASCII, в строке Python 3.X?
6. Каковы основные отличия между файлами в текстовом и двоичном режимах в Python 3.X?
7. Как можно прочитать текстовый файл Unicode, который содержит текст в кодировке, отличающейся от стандартной для имеющейся платформы?
8. Как можно создать текстовый файл Unicode в формате специфической кодировки?
9. Почему текст ASCII считается разновидностью текста Unicode?
10. Насколько большое влияние оказывает на имеющийся код изменение строковых типов в Python 3.X?

Проверьте свои знания: ответы

1. В Python 3.X есть три строковых типа: `str` (для текста Unicode, включая ASCII), `bytes` (для двоичных данных с абсолютными значениями байтов) и `bytearray` (изменяемый вариант `bytes`). Тип `str` обычно представляет содержимое, хранящееся в текстовом файле, а два других типа, как правило, представляют содержимое, которое хранится в двоичных файлах.
2. В Python 2.X есть два основных строковых типа: `str` (для 8-битных текстовых и двоичных данных) и `unicode` (для текста Unicode с возможно более широкими символами). Тип `str` применяется для содержимого текстовых и двоичных файлов; `unicode` используется для содержимого текстовых файлов, которое в целом сложнее 8-битных символов. В Python 2.6 (но не в более ранних версиях) также присутствует тип `bytearray` из Python 3.X, но по большей части он является обратным переносом и не демонстрирует настолько четкого различения текстовый/двоичный, как в Python 3.X.
3. Строковые типы Python 2.X не соответствуют напрямую строковым типам Python 3.X, поскольку тип `str` из Python 2.X приравнивается к типам `str` и `bytes` в Python 3.X, а тип `str` из Python 3.X – к типам `str` и `unicode` в Python 2.X. Изменяемость типа `bytearray` в Python 3.X тоже уникальна. Однако в целом текст Unicode обрабатывается типами `str` в Python 3.X и `unicode` в Python 2.X, байтовые данные поддерживаются типами `bytes` в Python 3.X и `str` в Python 2.X, а типы `bytes` в Python 3.X и `str` в Python 2.X обрабатывают более простые виды текста.
4. Строковые типы Python 3.X разделяют почти все те же самые операции: вызовы методов, операции над последовательностями и даже более крупные инструменты наподобие сопоставления с образцом работают одинаково. С другой стороны, только тип `str` поддерживает операции строкового форматирования,

а тип bytearray снабжен дополнительным набором операций, которые обеспечивают изменение на месте. Типы str и bytes также имеют методы соответственно для кодирования и декодирования текста.

5. Символы Unicode, отличные от ASCII, могут записываться в строке с помощью шестнадцатеричных управляющих последовательностей (\xNN) и управляющих последовательностей Unicode (\uNNNN, \UNNNNNNNN). На ряде компьютеров некоторые символы не ASCII – скажем, определенные символы Latin-1 – могут также набираться или вставляться прямо в код, и они интерпретируются в соответствии со стандартной кодировкой UTF-8 либо комментарием с директивой кодировки в исходном коде.

6. В Python 3.X файлы в текстовом режиме предполагают, что их содержимое является текстом Unicode (даже если все символы относятся к ASCII), и выполняют автоматическое декодирование при чтении и кодирование при записи. В случае файлов в двоичном режиме байты передаются в файл и обратно без изменений. Содержимое файлов в текстовом режиме обычно представляется в сценарии как объекты str, а содержимое файлов в двоичном режиме – как объекты bytes (или bytearray). Файлы в текстовом режиме также обрабатывают маркер BOM для кодировок определенных видов и при вводе и выводе автоматически транслируют последовательности конца строки в и из одиночного символа \n, если только такая трансляция явно не отключена; файлы в двоичном режиме не делают ничего из этого. В Python 2.X для файлов Unicode применяется вызов codecs.open, который выполняет кодирование и декодирование похожим образом; вызов open из Python 2.X транслирует символы конца строки только в текстовом режиме.

7. Чтобы прочитать файл, представленный в кодировке, которая отличается от стандартной для вашей платформы, просто передайте имя кодировки файла встроенной функции open в Python 3.X (codecs.open() в Python 2.X); данные будут декодироваться в соответствии с указанной кодировкой во время чтения из файла. Вы также можете читать в двоичном режиме и вручную декодировать байты в строку, указывая имя кодировки, но такой подход сопряжен с дополнительной работой и отчасти подвержен ошибкам в случае многобайтовых символов (вы можете непредумышленно прочитать неполную последовательность символа).

8. Чтобы создать файл с текстом Unicode в формате специфической кодировки, передайте имя желательной кодировки вызову open в Python 3.X (codecs.open() в Python 2.X); строки будут кодироваться в соответствии с указанной кодировкой при записи в файл. Вы также можете вручную закодировать строку в байты и записать ее в двоичном режиме, но обычно это будет дополнительной работой.

9. Текст ASCII считается разновидностью текста Unicode из-за того, что его 7-битный диапазон значений является подмножеством большинства кодировок Unicode. Например, допустимый текст ASCII будет также допустимым текстом Latin-1 (Latin-1 просто назначает оставшимся значениям в 8-битном байте добавочные символы) и допустимым текстом UTF-8 (UTF-8 определяет схему с переменным количеством байтов для представления большего числа символов, но символы ASCII по-прежнему представляются с теми же самыми кодами в единственном байте). Это делает кодировку Unicode обратно совместимой с массой

текстовых данных ASCII во всем мире (хотя она также может иметь ограниченные возможности – например, представление самоидентифицируемого текста может быть затруднено (правда, маркеры BOM исполняют во многом такую роль)).

10. Влияние изменения строковых типов в Python 3.X зависит от видов используемых строк. В отношении сценариев, где применяется простой текст ASCII на платформах со стандартными кодировками, совместимыми с ASCII, влияние, по всей видимости, будет незначительным: в этом случае строковый тип `str` работает одинаково в Python 2.X и 3.X. Более того, хотя связанные со строками инструменты в стандартной библиотеке, подобные `re`, `struct`, `pickle` и `xml`, формально могут использовать в Python 3.X не такие типы, как в Python 2.X, изменения практически несущественны для большинства программ, потому что типы `str` и `bytes` из Python 3.X и тип `str` из Python 2.X поддерживают почти идентичные интерфейсы. Если вы обрабатываете данные Unicode, то необходимый инструментальный набор просто перекочевал из `unicode` и `codecs.open()` версии Python 2.X в `str` и `open` версии.Python 3.X. Если вы имеете дело с файлами двоичных данных, тогда вам придется работать с содержимым как с объектами `bytes`; тем не менее, поскольку они имеют интерфейс, сходный со строками Python 2.X, влияние снова должно быть минимальным. Однако, как выясняется, есть много случаев, когда обязательный статус Unicode в Python 3.X подразумевает изменения в API-интерфейсах стандартной библиотеки – от работы с сетью и графическими пользовательскими интерфейсами до взаимодействия с базами данных и электронной почтой. Вообще говоря, Unicode вероятно со временем повлияет на большинство пользователей Python 3.X.

ГЛАВА 38

Управляемые атрибуты

В этой главе расширяется ранее представленная методика *перехвата атрибутов* и вводится еще одна, после чего они используются в нескольких более крупных примерах. Подобно всему остальному в данной части книги глава позиционируется, как посвященная сложным темам, и предназначена для факультативного чтения, поскольку большинству программистов приложений не нужно беспокоиться об обсуждаемых здесь материалах – они могут извлекать и устанавливать атрибуты объектов, не заботясь о реализации атрибутов.

Однако в особенности для разработчиков инструментов управление доступом к атрибутам может быть важной частью гибких API-интерфейсов. Кроме того, понимание раскрываемой в главе дескрипторной модели может сделать связанные инструменты, такие как слоты и свойства, более ясными и даже стать обязательным для изучения, если они встречаются в коде, с которым вы должны работать.

Для чего используются управляемые атрибуты?

Атрибуты объектов являются центральной частью большинства программ на Python – именно в них мы часто храним информацию о сущностях, обрабатываемых сценариями. Обычно атрибуты представляют собой просто имена для объектов; скажем, атрибут name лица может быть строкой, извлекаемой и устанавливаемой с помощью базового синтаксиса атрибутов:

```
person.name                 # Извлечь значение атрибута
person.name = значение      # Изменить значение атрибута
```

В большинстве случаев атрибуты находятся в самом объекте или наследуются из класса, от которого произведен объект. Такой базовой модели достаточно для большей части программ, которые вам доведется писать на протяжении вашей карьеры как программиста на Python.

Тем не менее, иногда требуется обеспечить высокую гибкость. Предположим, что вы пишете программу для использования атрибута name напрямую, но затем требования меняются – например, вы решаете, что имена должны проверяться посредством логики при установке или каким-то образом видоизменяться при извлечении. Реализовать методы для управления доступом к значению атрибута довольно легко (valid и transform здесь абстрактны):

```
class Person:
    def getName(self):
        if not valid():
            raise TypeError('cannot fetch name')     # не удается извлечь имя
        else:
            return self.name.transform()

    def setName(self, value):
        if not valid(value):
            raise TypeError('cannot change name')    # не удается изменить имя
        else:
            self.name = transform(value)

person = Person()
person.getName()
person.setName('value')
```

Однако это также требует внесения изменений во всех местах целой программы, где применяются имена – вероятно нетривиальная задача. Вдобавок такой подход требует, чтобы программе было известно, каким образом экспортируются значения: как простые имена или как вызываемые методы. Если вы начинали с интерфейса к данным на основе методов, тогда клиенты невосприимчивы к изменениям; если же нет, то ситуация может стать проблематичной.

Подобная проблема способна возникать чаще, чем ожидается. Скажем, значение ячейки в программе для работы с электронными таблицами может начать свое существование как простая дискретная величина, но позже видоизмениться в произвольное вычисление. Так как интерфейс объекта должен быть достаточно гибким, чтобы поддерживать такие изменения в будущем, не нарушая работу существующего кода, переключение на методы в более позднее время далеко от идеала.

Вставка кода для запуска при доступе к атрибутам

Более удачное решение позволило бы запускать код автоматически при доступе к атрибутам, когда есть необходимость. В том и заключается одна из главных ролей управляемых атрибутов – они предоставляют способы добавления логики *средства доступа к атрибутам* после их создания. В более общем случае они поддерживают произвольные режимы использования атрибутов, которые выходят за рамки простого хранения данных.

В различных местах книги нам встречались инструменты Python, которые позволяют сценариям динамически вычислять значения атрибутов при их извлечении и проверять либо изменять значения атрибутов при их сохранении. В этой главе мы собираемся раскрыть уже представленные инструменты, исследовать другие доступные инструменты и изучить ряд более крупных примеров применения в данной предметной области. В частности, в главе рассматриваются *четыре* методики доступа.

- Методы `__getattr__` и `__setattr__`, предназначенные для направления операций извлечения неопределенных атрибутов и операций присваивания всех атрибутов обобщенным методам обработчиков.
- Метод `__getattribute__`, предназначенный для направления операций извлечения всех атрибутов обобщенному методу обработчика.
- Встроенная функция `property`, предназначенная для направления операции доступа к конкретному атрибуту функциям обработчиков получения и установки.

- Дескрипторный протокол, предназначенный для направления операций доступа к конкретному атрибуту экземплярам классов с произвольными методами обработчиков получения и установки и являющийся основой для других инструментов, таких как свойства и слоты.

Инструменты в первом пункте списка доступны во всех версиях Python, а в следующих трех пунктах — в Python 3.X и классах нового стиля в Python 2.X. Они впервые появились в Python 2.2 вместе со многими другими расширенными инструментами, описанными в главе 32, такими как слоты и `super`. Инструменты из первого и третьего пунктов кратко упоминались в главах 30 и 32, а инструменты из второго и четвертого пунктов — по большому счету новые темы, которые мы исследуем в настоящей главе.

Вы увидите, что все четыре методики в известной мере разделяют цели, поэтому заданную задачу обычно можно решить с использованием любой из них. Тем не менее, они имеют важные отличия. Например, последние две методики применяются к *конкретным* атрибутам, тогда как первые две являются достаточно обобщенными, чтобы использоваться основанными на делегировании промежуточными классами, которые обязаны направлять внутренним объектам запросы *произвольных* атрибутов. Как будет показано далее, все четыре схемы также отличаются с точки зрения сложности и эстетики, о чем вы должны судить самостоятельно, посмотрев на них в действии.

Помимо изучения особенностей четырех методик перехвата атрибутов, перечисленных ваше, в настоящей главе также предоставляется возможность исследования более сложных программ, чем можно было видеть в других местах книги. Скажем, учебный пример `CardHolder` в конце главы послужит иллюстрацией крупных классов в действии. Некоторые из обрисованных здесь методик мы будем применять также в следующей главе при написании декораторов, поэтому прежде чем двигаться дальше, удостоверьтесь в том, что хотя бы в целом понимаете темы, рассматриваемые в текущей главе.

Свойства

Протокол свойств позволяет направлять операции извлечения, установки и удаления конкретного атрибута предоставляемым нами функциям или методам, что дает возможность вставлять код, подлежащий автоматическому выполнению при доступе к атрибуту, перехватывать операции удаления атрибута и при желании обеспечивать документацией по атрибутам.

Свойства создаются с помощью встроенной функции `property` и присваиваются атрибутам класса в точности как функции методов. Соответственно, подобно любым другим атрибутам класса они наследуются подклассами и экземплярами. Их функции перехвата доступа снабжаются аргументом экземпляра `self`, который дает доступ к информации состояния и атрибутам класса, доступным в переданном экземпляре.

Свойство управляет одним конкретным атрибутом. Хотя свойство не может обобщенно перехватывать все операции доступа к атрибуту, оно позволяет контролировать операции извлечения и присваивания и свободно делать вычисляемым атрибут, представляющий собой хранилище простых данных; работа существующего кода при этом не нарушается. Вы увидите, что свойства тесно связаны с дескрипторами; по существу они считаются их ограниченной формой.

Основы

Свойство создается присваиванием результата вызова встроенной функции атрибуту класса:

```
атрибут = property(fget, fset, fdel, doc)
```

Ни один из аргументов встроенной функции property не является обязательным, и все они получают стандартное значение None, если не передается иное. Для первых трех аргументов None означает, что соответствующая операция не поддерживается, и попытка ее выполнить приведет к генерации исключения AttributeError.

Когда мы используем аргументы property, то передаем в fget функцию для перехвата операций извлечения атрибута, в fset функцию для перехвата операций присваивания и в fdel функцию для перехвата операций удаления. Формально все три аргумента принимают любой вызываемый объект, в том числе метод класса, имеющий первый аргумент для получения уточняемого экземпляра. При последующем вызове функция fget возвращает вычисленное значение атрибута, fset и fdel ничего не возвращают (на самом деле None) и все три могут генерировать исключения, чтобы отклонять запросы на доступ.

Аргумент doc принимает строку документации для атрибута, если ее наличие желательно; в противном случае свойство копирует строку документации функции fget, которая обычно по умолчанию установлена в None.

Вызов встроенной функции property возвращает объект свойства, который мы присваиваем имени атрибута, подлежащего управлению в области видимости класса, где он будет наследоваться каждым экземпляром.

Первый пример

Чтобы показать, как все выглядит в коде, в следующем классе применяется свойство для отслеживания доступа к атрибуту по имени name; действительные хранящиеся данные называются _name, так что они не конфликтуют со свойством:

```
class Person:                       # Добавить (object) в Python 2.X
    def __init__(self, name):
        self._name = name
    def getName(self):
        print('fetch...')           # извлечение...
        return self._name
    def setName(self, value):
        print('change...')          # изменение...
        self._name = value
    def delName(self):
        print('remove...')          # удаление...
        del self._name
    name = property(getName, setName, delName, "name property docs")
                                               # документация по свойству name
bob = Person('Bob Smith')           # Экземпляр bob имеет управляемый атрибут
print(bob.name)                     # Запускается getName
bob.name = 'Robert Smith'           # Запускается setName
print(bob.name)
del bob.name                        # Запускается delName

print('-'*20)
sue = Person('Sue Jones')           # Экземпляр sue тоже наследует свойство
print(sue.name)
print(Person.name.__doc__)          # Или help(Person.name)
```

Свойства доступны в Python 2.X и 3.X, но для корректной работы с операциями *присваивания* в Python 2.X требуют наследования нового стиля от `object` — чтобы выполнить код в Python 2.X, добавьте `object` в качестве суперкласса. Указывать суперкласс `object` можно и в Python 3.X, но он подразумевается и не требуется, а потому временами в книге опущен ради снижения перегруженности.

Это конкретное свойство мало что делает (оно просто перехватывает операции доступа и отслеживает атрибут), но предназначено для демонстрации самого протокола. После запуска кода два экземпляра наследуют свойство, как поступили бы с любым другим атрибутом, присоединенным к их классу. Однако операции доступа к атрибуту перехватываются:

```
c:\code> py -3 prop-person.py
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
name property docs
```

Как и все атрибуты класса, свойства *наследуются* экземплярами и подклассами на более низких уровнях. Например, если мы изменим код примера:

```
class Super:
    ...первоначальный код класса Person...
    name = property(getName, setName, delName, 'name property docs')

class Person(Super):
    pass                # Свойства наследуются (подобно всем атрибутам класса)

bob = Person('Bob Smith')
...остальной код остался прежним...
```

то вывод останется тем же самым — подкласс `Person` наследует свойство `name` от `Super`, а экземпляр `bob` получает его из `Person`. С точки зрения наследования свойства работают аналогично нормальным методам; поскольку они имеют доступ к аргументу экземпляра `self`, то могут обращаться к информации состояния и методам независимо от того, насколько глубоко находится подкласс, как в дальнейшем иллюстрируется в следующем разделе.

Вычисляемые атрибуты

В примере из предыдущего раздела было реализовано просто отслеживание доступа к атрибуту. Тем не менее, обычно свойства способны делать гораздо большее — скажем, динамически вычислять значение атрибута при извлечении, как показано ниже:

```
class PropSquare:
    def __init__(self, start):
        self.value = start
    def getX(self):                     # При извлечении атрибута
        return self.value ** 2
    def setX(self, value):              # При присваивании атрибута
        self.value = value
    X = property(getX, setX)            # Удаления и документации не предусмотрено
```

```
P = PropSquare(3)              # Два экземпляра класса со свойством
Q = PropSquare(32)             # Каждый имеет отличающуюся информацию состояния

print(P.X)                     # 3 ** 2
P.X = 4
print(P.X)                     # 4 ** 2
print(Q.X)                     # 32 ** 2 (1024)
```

В классе PropSquare определен атрибут X, доступ к которому осуществляется так, как если бы он был статическими данными, но на самом деле при извлечении атрибута X запускается код для вычисления его значения. Эффект во многом похож на неявный вызов метода. Во время выполнения кода значение сохраняется в экземпляре как информация состояния, но каждый раз, когда мы извлекаем его через управляемый атрибут, значение автоматически возводится в квадрат:

```
c:\code> py -3 prop-computed.py
9
16
1024
```

Обратите внимание, что мы создали два разных экземпляра — поскольку методы свойства автоматически принимают аргумент self, они имеют доступ к информации состояния, хранящейся в экземплярах. В рассматриваемом случае это означает, что при извлечении вычисляется квадрат собственных данных переданного в self экземпляра.

Реализация свойств с помощью декораторов

Хотя мы откладываем исследование дополнительных деталей до следующей главы, основы *декораторов функций* были представлены ранее в главе 32. Вспомните, что синтаксис декораторов функций:

```
@decorator
def func(args): ...
```

автоматически транслируется интерпретатором Python в показанный ниже эквивалент для повторной привязки имени функции к результату, который возвращает вызываемый объект decorator:

```
def func(args): ...
func = decorator(func)
```

Из-за такого отображения оказывается, что встроенная функция property может использоваться в качестве декоратора для определения функции, которая будет запускаться автоматически при извлечении атрибута:

```
class Person:
    @property
    def name(self): ...    # Повторная привязка: name = property(name)
```

Во время выполнения декорированный метод автоматически передается в первом аргументе встроенной функции property. На самом деле декораторы функций являются всего лишь альтернативным синтаксисом для создания свойства и повторной привязки имени атрибута вручную, но в такой роли его можно считать более явным подходом:

```
class Person:
    def name(self): ...
    name = property(name)
```

Декораторы для установки и удаления

Начиная с версий Python 2.6 и 3.0, объекты свойств также имеют методы getter, setter и deleter, которые назначают соответствующие методы доступа к свойству и возвращают копию самого свойства. Мы можем применять их для указания компонентов свойств, также декорируя нормальные методы, хотя компонент getter обычно заполняется автоматически самим процессом создания свойства:

```
class Person:
   def __init__(self, name):
      self._name = name

@property
def name(self):                    # name = property(name)
   "name property docs"            # документация по свойству name
   print('fetch...')               # извлечение...
   return self._name

@name.setter
def name(self, value):             # name = name.setter(name)
   print('change...')              # изменение...
   self._name = value

@name.deleter
def name(self):                    # name = name.deleter(name)
   print('remove...')              # удаление...
   del self._name

bob = Person('Bob Smith')          # Экземпляр bob имеет управляемый атрибут
print(bob.name)                    # Запускается метод getter свойства name (name 1)
bob.name = 'Robert Smith'          # Запускается метод setter свойства name (name 2)
print(bob.name)
del bob.name                       # Запускается метод deleter свойства name (name 3)

print('-'*20)
sue = Person('Sue Jones')          # Экземпляр sue тоже наследует свойство
print(sue.name)
print(Person.name.__doc__)         # Или help(Person.name)
```

Приведенный код фактически эквивалентен первому примеру в настоящем разделе — декорирование в данном случае представляет собой лишь альтернативный способ реализации свойств. Запуск кода дает те же самые результаты:

```
c:\code> py -3 prop-person-deco.py
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
name property docs
```

По сравнению с ручным присваиванием результатов property использование декораторов для реализации свойств требует только трех дополнительных строк кода — разница, кажущаяся пренебрежимо малой. Однако, как часто бывает с альтернативными инструментами, выбор между двумя методиками по большей части субъективен.

Дескрипторы

Дескрипторы предлагают альтернативный способ перехвата доступа к атрибутам; они тесно связаны со свойствами, которые обсуждались в предыдущем разделе. В действительности свойство *является* разновидностью дескриптора — говоря формально, встроенная функция `property` представляет собой всего лишь упрощенный способ создания особого типа дескриптора, который запускает функции методов при доступе к атрибуту. По сути, дескрипторы — это внутренний механизм реализации для разнообразных инструментов, относящихся к классам, в том числе свойств и слотов.

С функциональной точки зрения дескрипторный протокол позволяет направлять операции извлечения, установки и удаления конкретного атрибута методам объекта экземпляра отдельного класса, которые мы предоставляем. В итоге мы получаем возможность вставлять код, подлежащий автоматическому запуску во время операций извлечения и присваивания, перехватывать операции удаления атрибутов и при желании снабжать документацией по атрибутам.

Дескрипторы создаются в виде независимых *классов* и присваиваются атрибутам классов в точности как функции методов. Подобно любым другим атрибутам классов они наследуются подклассами и экземплярами. Их методы перехвата доступа снабжаются как `self` для экземпляра самого дескриптора, так и экземпляром клиентского класса, чей атрибут ссылается на объект дескриптора. По указанной причине они могут предохранять собственную информацию состояния, а также информацию состояния экземпляра клиентского класса. Например, дескриптор может вызывать методы, доступные в клиентском классе, плюс определенные в нем методы, специфичные для дескриптора.

Как и свойство, дескриптор управляет одним конкретным атрибутом. Хотя дескриптор не способен перехватывать все операции доступа к атрибуту обобщенным образом, он обеспечивает контроль над операциями извлечения и присваивания и позволяет свободно изменять имя атрибута с простого хранилища данных на вычисляемый атрибут, не нарушая работу существующего кода. Свойства на самом деле являются просто удобным способом создания особого вида дескриптора и, как мы увидим, они могут быть непосредственно реализованы как дескрипторы.

В отличие от свойств дескрипторы охватывают более широкие границы и предоставляют более универсальный инструмент. Скажем, из-за того, что дескрипторы реализуются как нормальные классы, они имеют собственное состояние, способны принимать участие в иерархиях наследования дескрипторов, могут применять композицию для агрегирования объектов и предлагают естественную структуру для написания кода внутренних методов и строк документации по атрибутам.

Основы

Как уже упоминалось, дескрипторы реализуются как отдельные классы и предоставляют особым образом именованные методы доступа для операций доступа к атрибутам, которые они перехватывают. Методы извлечения, установки и удаления автоматически запускаются, когда в отношении атрибута, которому присвоен экземпляр класса дескриптора, выполняется соответствующая операция доступа:

```
class Descriptor:
    "docstring goes here"                      # Строка документации
    def __get__(self, instance, owner): ...   # Возвращает значение атрибута
    def __set__(self, instance, value): ...   # Ничего не возвращает (None)
    def __delete__(self, instance): ...       # Ничего не возвращает (None)
```

Классы с любым таким методом считаются дескрипторами, а их методы будут специальными, когда один из их экземпляров присваивается атрибуту другого класса – при доступе к атрибуту методы автоматически вызываются. Если любой из методов отсутствует, то обычно это значит, что соответствующий вид доступа не поддерживается. Тем не менее, в отличие от свойств пропуск `__set__` позволяет присваивать значение имени атрибута дескриптора и, следовательно, переопределять его в экземпляре, тем самым *скрывая* дескриптор – чтобы сделать атрибут допускающим *только чтение*, потребуется определить `__set__` для перехвата операций присваивания и генерации исключения.

Дескрипторы с методами `__set__` также имеют последствия в особых случаях в отношении наследования, рассмотрение которых мы в основном отложим до главы 40, где будут раскрыты метаклассы и дана полная спецификация наследования. Вкратце дескриптор с методом `__set__` формально известен как *дескриптор данных* и получает преимущество перед другими именами, которые ищутся согласно нормальным правилам наследования. Например, унаследованный дескриптор для имени `__class__` переопределяет то же самое имя в словаре пространств имен экземпляра. Это также помогает гарантировать, что дескрипторы данных, реализуемые вами в собственных классах, имеют преимущество перед остальными.

Аргументы методов дескриптора

Прежде чем мы напишем какой-то реалистичный код, давайте кратко ознакомимся с основами. Всем трем методам дескриптора, обрисованным в предыдущем разделе, передаются экземпляр класса дескриптора (`self`) и экземпляр клиентского класса, к которому присоединен экземпляр дескриптора (`instance`).

Метод доступа `__get__` вдобавок принимает аргумент, указывающий класс, к которому присоединен экземпляр дескриптора. Его аргумент `instance` представляет собой либо экземпляр, через который осуществлялся доступ к атрибуту (для *`instance`*`.attr`), либо `None`, когда доступ к атрибуту производился напрямую через класс владельца (для *`class`*`.attr`). Первый вариант, как правило, вычисляет значение для операции доступа через экземпляр, а второй обычно возвращает `self`, если поддерживается доступ к объекту дескриптора.

Скажем, в приведенном ниже сеансе Python 3.X при извлечении `X.attr` интерпретатор Python автоматически запускает метод `__get__` экземпляра класса `Descriptor`, который был присвоен атрибуту класса `Subject.attr`. В Python 2.X используйте эквивалент оператора `print` и унаследуйте *оба* класса от `object`, т.к. дескрипторы являются инструментом классов нового стиля; в Python 3.X такое наследование подразумевается и может быть опущено, хотя его наличие вреда не причинит:

```
>>> class Descriptor:                        # Добавить (object) в Python 2.X
        def __get__(self, instance, owner):
            print(self, instance, owner, sep='\n')
>>> class Subject:                           # Добавить (object) в Python 2.X
        attr = Descriptor()                  # Экземпляр Descriptor - атрибут класса

>>> X = Subject()
>>> X.attr
<__main__.Descriptor object at 0x0281E690>
<__main__.Subject object at 0x028289B0>
<class '__main__.Subject'>

>>> Subject.attr
<__main__.Descriptor object at 0x0281E690>
None
<class '__main__.Subject'>
```

Обратите внимание на аргументы, автоматически переданные методу `__get__` при первом извлечении атрибута — когда извлекается `X.attr`, то словно происходит следующая трансляция (хотя `Subject.attr` здесь не вызывает `__get__` еще раз):

```
X.attr -> Descriptor.__get__(Subject.attr, X, Subject)
```

Дескриптору известно, что к нему обращаются напрямую, если его аргументом экземпляра оказывается `None`.

Дескрипторы атрибутов только для чтения

Как упоминалось ранее, в отличие от свойств простого отсутствия метода `__set__` в дескрипторе недостаточно для того, чтобы сделать атрибут допускающим только чтение, т.к. имени дескриптора может быть присвоено значение в экземпляре. В следующем сеансе присваивание атрибуту `X.a` сохраняет `a` в объекте экземпляра `X`, тем самым скрывая дескриптор, который хранится в классе `C`:

```
>>> class D:
        def __get__(*args): print('get')
>>> class C:
        a = D()            # Атрибут a - это экземпляр дескриптора
>>> X = C()
>>> X.a                    # Запускается метод __get__ унаследованного дескриптора
get
>>> C.a
get
>>> X.a = 99               # Сохраняется в X, скрывая C.a!
>>> X.a
99
>>> list(X.__dict__.keys())
['a']
>>> Y = C()
>>> Y.a                    # Y по-прежнему наследует дескриптор
get
>>> C.a
get
```

Таким способом в Python работают все операции присваивания атрибутам экземпляров, что позволяет классам избирательно переопределять стандартные установки уровня классов в их экземплярах. Чтобы сделать основанный на дескрипторе атрибут допускающим только чтение, перехватывайте операцию присваивания в классе дескриптора и генерируйте исключение для предотвращения присваивания атрибуту. Когда выполняется присваивание атрибуту, являющемуся дескриптором, интерпретатор Python фактически обходит нормальное поведение присваивания на уровне экземпляров и направляет операцию объекту дескриптора:

```
>>> class D:
        def __get__(*args): print('get')
        def __set__(*args): raise AttributeError('cannot set')
                                                          # Не может быть установлен
>>> class C:
        a = D()
>>> X = C()
>>> X.a                    # Направляется C.a.__get__
get
>>> X.a = 99               # Направляется C.a.__set__
AttributeError: cannot set
Ошибка атрибута: не может быть установлен
```

Также будьте осторожны, чтобы не спутать метод `__delete__` дескриптора с универсальным методом `__del__`. Первый вызывается при попытках удалить имя управляемого атрибута из экземпляра класса владельца; второй представляет собой универсальный метод деструктора экземпляра, запускаемый в то время, когда экземпляр любого класса подвергается сборке мусора. Метод `__delete__` более тесно связан с обобщенным методом удаления атрибутов `__delattr__`, который мы встретим позже в главе. Методы перегрузки операций подробно обсуждались в главе 30.

Первый пример

Чтобы посмотреть, как все объединяется в более реалистичном примере, мы начнем с того же первого примера, который был написан для свойств. Ниже определяется дескриптор, который перехватывает доступ к атрибуту по имени `name` в своих клиентах. Его методы применяют аргумент `instance` для доступа к информации состояния в передаваемом экземпляре, где фактически хранится строка имени. Подобно свойствам дескрипторы работают надлежащим образом только с классами нового стиля, поэтому если вы используете Python 2.X , то обязательно унаследуйте от `object` *оба* класса – недостаточно унаследовать только дескриптор или только его клиент:

```
class Name:                              # Добавить (object) в Python 2.X
    "name descriptor docs"               # документация по свойству name
    def __get__(self, instance, owner):
        print('fetch...')                # извлечение...
        return instance._name
    def __set__(self, instance, value):
        print('change...')               # изменение...
        instance._name = value
    def __delete__(self, instance):
        print('remove...')               # удаление...
        del instance._name

class Person:                            # Добавить (object) в Python 2.X
    def __init__(self, name):
        self._name = name
    name = Name()                        # Присвоить дескриптор атрибуту

bob = Person('Bob Smith')                # Экземпляр bob имеет управляемый атрибут
print(bob.name)                          # Запускается Name.__get__
bob.name = 'Robert Smith'                # Запускается Name.__set__
print(bob.name)
del bob.name                             # Запускается Name.__delete__

print('-'*20)
sue = Person('Sue Jones')                # Экземпляр sue тоже наследует дескриптор
print(sue.name)
print(Name.__doc__)                      # Или help(Name)
```

Обратите внимание в коде на то, что мы присваиваем экземпляр класса дескриптора *атрибуту класса* в клиентском классе; благодаря этому он наследуется всеми экземплярами класса в точности как методы класса. Вообще говоря, мы *обязаны* присваивать дескриптор атрибуту класса – дескриптор не будет работать, если взамен присвоить его атрибуту экземпляра `self`. При запуске методу `__get__` дескриптора передаются три объекта для определения его контекста:

- `self` — экземпляр класса `Name`;
- `instance` — экземпляр класса `Person`;
- `owner` — класс `Person`.

Во время выполнения этого кода методы дескриптора перехватывают операции доступа к атрибуту во многом подобно версии со свойством. В действительности вывод оказывается снова таким же:

```
c:\code> py -3 desc-person.py
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
name descriptor docs
```

Также подобно примеру со свойством экземпляр класса дескриптора является атрибутом класса и потому наследуется всеми экземплярами клиентского класса и его подклассов. Скажем, если мы изменим класс `Person` в примере следующим образом, то вывод сценария останется тем же самым:

```
...
class Super:
    def __init__(self, name):
        self._name = name
    name = Name()

class Person(Super):                    # Дескрипторы наследуются
                                        #   (т.к. являются атрибутами класса)
    pass
...
```

Кроме того, когда класс дескриптора бесполезен за рамками клиентского класса, то вполне разумно синтаксически внедрить определение дескриптора в его клиент. Вот как наш пример выглядит в случае применения *вложенного класса*:

```
class Person:
    def __init__(self, name):
        self._name = name

class Name:                             # Использование вложенного класса
    "name descriptor docs"
    def __get__(self, instance, owner):
        print('fetch...')
        return instance._name
    def __set__(self, instance, value):
        print('change...')
        instance._name = value
    def __delete__(self, instance):
        print('remove...')
        del instance._name
    name = Name()
```

При такой реализации Name становится локальной переменной в области видимости, принадлежащей оператору определения класса Person, и не будет конфликтовать с любыми именами вне класса. Данная версия работает аналогично первоначальной версии – мы просто перенесли определение класса дескриптора внутрь области видимости клиентского класса, – но последнюю строку тестового кода потребуется изменить, чтобы извлекать строку документации из нового местоположения (файл desc-person-nested.py):

```
...
print(Person.Name.__doc__)           # Отличие: не Name.__doc__ вне класса
```

Вычисляемые атрибуты

Как и в случае использования свойств, наш первый пример дескриптора из предыдущего раздела делал не особо многое – он всего лишь выводил трассировочные сообщения по мере доступа к атрибутам. На практике дескрипторы могут также применяться для вычисления значений атрибутов при каждом их извлечении. Сказанное иллюстрируется ниже в переделанном примере со свойством, где теперь используется дескриптор для автоматического возведения в квадрат значения атрибута каждый раз, когда оно извлекается:

```
class DescSquare:
    def __init__(self, start):            # Каждый дескриптор имеет
                                          #   собственное состояние
        self.value = start
    def __get__(self, instance, owner):   # При извлечении атрибута
        return self.value ** 2
    def __set__(self, instance, value):   # При присваивании атрибута
        self.value = value                # Операция удаления и строка
                                          #   документации отсутствуют

class Client1:
    X = DescSquare(3)        # Присвоить экземпляр дескриптора атрибуту класса

class Client2:
    X = DescSquare(32)       # Еще один экземпляр в другом клиентском классе
                             # Можно было бы также предусмотреть
                             #   два экземпляра в том же самом классе

c1 = Client1()
c2 = Client2()

print(c1.X)                  # 3 ** 2
c1.X = 4
print(c1.X)                  # 4 ** 2
print(c2.X)                  # 32 ** 2 (1024)
```

Вывод, полученный в результате запуска примера, будет таким же, как в первоначальной версии со свойством, но здесь операции доступа к атрибуту перехватываются объектом класса дескриптора:

```
c:\code> py -3 desc-computed.py
9
16
1024
```

Использование информации состояния в дескрипторах

Изучив реализованные до сих пор два примера с дескрипторами, вы можете заметить, что они получают свою информацию из разных мест — первый (пример с атрибутом name) работает с данными, хранящимися в *экземпляре* клиентского класса, а второй (пример с возведением в квадрат значения атрибута) задействует данные, присоединенные к самому объекту *дескриптора* (self). Фактически дескрипторы могут использовать состояние экземпляра *и* состояние дескриптора, а также любое их сочетание.

- Состояние дескриптора применяется для управления либо данными, используемыми при внутренней работе дескриптора, либо данными, которые охватывают все экземпляры. Оно может варьироваться в зависимости от места появления атрибута (часто в зависимости от клиентского класса).
- Состояние экземпляра хранит информацию, связанную и возможно созданную клиентским классом. Оно может варьироваться в зависимости от экземпляра клиентского класса (т.е. в зависимости от объекта приложения).

Другими словами, состояние дескриптора представляет собой данные для каждого дескриптора, а состояние экземпляра — данные для каждого экземпляра клиента. Как принято в ООП, вы обязаны тщательно выбирать состояние. Например, обычно вы не должны применять состояние *дескриптора* для записи имен сотрудников, поскольку каждый экземпляр клиента требует собственного значения — если они хранятся в дескрипторе, то каждый экземпляр клиентского класса будет фактически разделять ту же самую одиночную копию. С другой стороны, как правило, вы не будете использовать состояние *экземпляра* для записи данных, относящихся к внутренней реализации дескриптора — если они хранятся в каждом экземпляре, тогда окажется множество разных копий.

Методы дескриптора могут применять любую из двух форм состояния, но состояние дескриптора часто делает ненужным использование специальных соглашений по именованию с целью избегания конфликтов имен в экземпляре для данных, которые не специфичны к экземпляру. Скажем, следующий дескриптор присоединяет информацию к собственному экземпляру, поэтому она не конфликтует с информацией в экземпляре клиентского класса, но и разделяет такую информацию между двумя экземплярами клиента:

```
class DescState:  # Использование состояния дескриптора, (object) в Python 2.X
    def __init__(self, value):
        self.value = value
    def __get__(self, instance, owner):       # При извлечении атрибута
        print('DescState get')
        return self.value * 10
    def __set__(self, instance, value):       # При присваивании атрибута
        print('DescState set')
        self.value = value

# Клиентский класс
class CalcAttrs:
    X = DescState(2)                          # Атрибут класса дескриптора
    Y = 3                                     # Атрибут класса
    def __init__(self):
        self.Z = 4                            # Атрибут экземпляра

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)          # X вычисляется, остальные нет
```

```
obj.X = 5                         # Присваивание X перехватывается
CalcAttrs.Y = 6                   # Y повторно присваивается в классе
obj.Z = 7                         # Z присваивается в экземпляре
print(obj.X, obj.Y, obj.Z)

obj2 = CalcAttrs()            # Но X использует разделяемые данные подобно Y!
print(obj2.X, obj2.Y, obj2.Z)
```

Информация о внутреннем атрибуте value существует только в *дескрипторе*, следовательно, в случае применения того же самого имени в экземпляре клиента конфликт не возникает. Обратите внимание, что здесь управляемым является только атрибут дескриптора – операции извлечения и установки X перехватываются, но операции доступа к Y и Z нет (атрибут Y присоединен к клиентскому классу, а Z к экземпляру). Во время выполнения приведенного выше кода атрибут X вычисляется при извлечении, но его значение также остается одинаковым для всех экземпляров клиента из-за использования состояния уровня дескриптора:

```
c:\code> py -3 desc-state-desc.py
DescState get
20 3 4
DescState set
DescState get
50 6 7
DescState get
50 6 4
```

Кроме того, для дескриптора вполне реально хранить или применять атрибут, присоединенный к *экземпляру* клиентского класса, а не к самому себе. Важно отметить, что в отличие от данных, хранящихся в самом дескрипторе, это делает возможными данные, которые способны варьироваться в зависимости от экземпляра клиентского класса. Дескриптор в показанном далее примере предполагает, что экземпляр имеет атрибут _X, присоединенный клиентским классом, и использует его для вычисления значения атрибута, который он представляет:

```
class InstState:  # Использование состояния экземпляра, (object) в Python 2.X
    def __get__(self, instance, owner):
      print('InstState get')  # Предполагается, что установлен клиентским классом
      return instance._X * 10
    def __set__(self, instance, value):
       print('InstState set')
       instance._X = value
# Клиентский класс
class CalcAttrs:
    X = InstState()                   # Атрибут класса дескриптора
    Y = 3                             # Атрибут класса
    def __init__(self):
        self._X = 2                   # Атрибут экземпляра
        self.Z = 4                    # Атрибут экземпляра
obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)            # X вычисляется, остальные нет
obj.X = 5                             # Присваивание X перехватывается
CalcAttrs.Y = 6                       # Y повторно присваивается в классе
obj.Z = 7                             # Z присваивается в экземпляре
print(obj.X, obj.Y, obj.Z)

obj2 = CalcAttrs()                    # Но X теперь отличается подобно Z!
print(obj2.X, obj2.Y, obj2.Z)
```

Как и ранее, X присваивается дескриптор, управляющий доступом. Однако новый дескриптор здесь не содержит информации, а применяет атрибут, который предположительно существует в экземпляре — во избежание конфликтов с именем самого дескриптора данный атрибут назван _X. Результаты выполнения этой версии похожи, но значение атрибута дескриптора может варьироваться в зависимости от экземпляра клиента из-за отличающейся политики состояния:

```
c:\code> py -3 desc-state-inst.py
InstState get
20 3 4
InstState set
InstState get
50 6 7
InstState get
20 6 4
```

С состоянием дескриптора и состоянием экземпляра связаны свои роли. На самом деле это и есть то общее преимущество, которым дескрипторы обладают по сравнению со свойствами — поскольку они имеют собственное состояние, то могут легко сохранять данные внутренне, не добавляя их к пространству имен в объекте экземпляра клиента. В качестве сводки в следующем дескрипторе используются *оба* источника состояния — в `self.data` хранится информация для каждого атрибута, тогда как `instance.data` может изменяться от экземпляра к экземпляру клиента:

```
>>> class DescBoth:
        def __init__(self, data):
            self.data = data
        def __get__(self, instance, owner):
            return '%s, %s' % (self.data, instance.data)
        def __set__(self, instance, value):
            instance.data = value

>>> class Client:
        def __init__(self, data):
            self.data = data
        managed = DescBoth('spam')

>>> I = Client('eggs')
>>> I.managed                       # Показать оба источника данных
'spam, eggs'
>>> I.managed = 'SPAM'              # Изменить данные экземпляра
>>> I.managed
'spam, SPAM'
```

Мы еще вернемся к последствиям выбора в более крупном учебном примере позже в главе. Прежде чем двигаться дальше, вспомните из обсуждения слотов в главе 32, что с помощью инструментов, подобных `dir` и `getattr`, мы можем получать доступ к "виртуальным" атрибутам вроде свойств и дескрипторов, хотя они не существуют в словаре пространств имен экземпляра. То, *должны* ли вы обращаться к ним таким способом, вероятно, зависит от программы — свойства и дескрипторы могут выполнять произвольные вычисления и быть менее очевидными "данными" экземпляра, чем слоты:

```
>>> I.__dict__
{'data': 'SPAM'}
>>> [x for x in dir(I) if not x.startswith('__')]
['data', 'managed']

>>> getattr(I, 'data')
```

```
'SPAM'
>>> getattr(I, 'managed')
'spam, SPAM'
>>> for attr in (x for x in dir(I) if not x.startswith('__')):
        print('%s => %s' % (attr, getattr(I, attr)))
data => SPAM
managed => spam, SPAM
```

Более обобщенные инструменты `__getattr__` и `__getattribute__`, которые мы встретим позже, не рассчитаны на поддержку такой функциональности – из-за того, что они не имеют атрибутов уровня класса, имена их "виртуальных" атрибутов не появляются в результатах `dir`[1]. Взамен они также не ограничиваются специфичными именами атрибутов, реализованных в виде свойств или дескрипторов: как объясняется в следующем разделе, данные инструменты разделяют даже больше, чем это поведение.

Связь между свойствами и дескрипторами

Как упоминалось ранее, свойства и дескрипторы тесно связаны – встроенная функция `property` является всего лишь удобным способом создания дескриптора. Теперь, когда вам известно, как работают оба средства, вы должны быть в состоянии видеть возможность эмуляции встроенной функции `property` с помощью класса дескриптора:

```
class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel                     # Сохранить несвязанный метод
        self.__doc__ = doc                   # или другие вызываемые объекты
    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("can't get attribute")  # нельзя извлечь атрибут
        return self.fget(instance)           # Передать instance экземпляру self
                                             # в методах доступа к свойствам
    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("can't set attribute") # нельзя установить атрибут
        self.fset(instance, value)
    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("can't delete attribute") # нельзя удалить атрибут
        self.fdel(instance)
class Person:
    def getName(self): print('getName...')
    def setName(self, value): print('setName...')
    name = Property(getName, setName)    # Использовать подобно property()
x = Person()
x.name
x.name = 'Bob'
del x.name
```

[1] Как отмечалось в главе 31, такие динамические классы могут также использовать метод `__dir__`, чтобы предоставлять результирующий список атрибутов для вызовов `dir`, хотя универсальные инструменты не могут полагаться на этот необязательный интерфейс.

Класс Property перехватывает операции доступа к атрибуту с помощью дескрипторного протокола и направляет запросы функциям или методам, которые были переданы и сохранены в состоянии дескриптора, когда класс создавался. Скажем, операции извлечения атрибута направляются из класса Person методу __get__ класса Property и обратно методу getName класса Person. Благодаря дескрипторам все "просто работает":

```
c:\code> py -3 prop-desc-equiv.py
getName...
setName...
AttributeError: can't delete attribute
Ошибка атрибута: нельзя удалить атрибут
```

Тем не менее, обратите внимание, что эквивалентный класс дескриптора Property обрабатывает только базовое применение свойств; чтобы использовать *декораторный синтаксис* @ также для спецификации операций установки и удаления, нам пришлось бы расширить класс Property методами setter и deleter, которые сохраняли бы декорированную функцию доступа и возвращали бы объект свойства (self должно быть достаточно). Так как встроенная функция property уже делает это, мы опустим здесь формальную реализацию такого расширения.

Дескрипторы, слоты и многое другое

Вероятно, теперь вы можете, по крайней мере, частично представить себе, как дескрипторы применяются для реализации расширения *слотов* в Python: словари атрибутов экземпляра аннулируются за счет создания дескрипторов уровня класса, которые перехватывают доступ к именам слотов и отображают такие имена на последовательное пространство хранения в экземпляре. Однако в отличие от вызова property б*о*льшая часть магии, связанной со слотами, организуется во время создания класса автоматически и неявно, когда в классе присутствует атрибут __slots__.

Дополнительные сведения о слотах ищите в главе 32 (там также изложены причины, почему их не рекомендуется использовать кроме как в патологических сценариях). Дескрипторы применяются и для других инструментов, связанных с классами, но мы не будем здесь приводить дальнейшие внутренние подробности; нужная информация доступна в руководствах и исходном коде Python.

В главе 39 мы будем использовать дескрипторы для реализации *декораторов* функций, которые применяются как к функциям, так и к методам. Вы увидите, что поскольку дескрипторы получают экземпляры дескрипторов *и* клиентских классов, они хороши в этой роли, хотя вложенные функции обычно обеспечивают концептуально гораздо более простое решение. В той же главе 39 мы еще задействуем дескрипторы как один из способов для перехвата извлечения методов *встроенных* операций.

Также обязательно ознакомьтесь с описанием приоритета *дескрипторов данных* в упоминавшейся ранее полной модели наследования: с помощью __set__ дескрипторы переопределяют другие имена и потому являются довольно связывающими — они не могут скрываться именами в словарях экземпляров.

__getattr__ и __getattribute__

До сих пор мы изучали свойства и дескрипторы – инструменты для управления конкретными атрибутами. Методы перегрузки операций `__getattr__` и `__getattribute__` предоставляют очередные способы перехвата извлечения атрибутов для экземпляров классов. Подобно свойствам и дескрипторам они позволяют вставлять код, подлежащий автоматическому запуску при доступе к атрибутам. Как вы увидите, указанные два метода могут также использоваться более универсально. Из-за того, что методы `__getattr__` и `__getattribute__` перехватывают доступ к произвольным именам, они применяются в более широких ролях вроде делегирования, но могут также влечь за собой дополнительные вызовы в ряде контекстом и являются чересчур динамическими, чтобы регистрироваться в результатах `dir`.

Перехват извлечения атрибутов имеет две разновидности, реализуемые посредством двух разных методов.

- Метод `__getattr__` запускается для неопределенных атрибутов – поскольку он выполняется только для атрибутов, которые не хранятся в экземпляре или не наследуются от одного из его классов, используется он прямолинейно.
- Метод `__getattribute__` запускается для каждого атрибута – поскольку он включает все, вы должны соблюдать осторожность при его применении, чтобы избежать рекурсивных циклов из-за передачи суперклассу операций доступа к атрибутам.

Первый метод встречался в главе 30; он доступен во всех версиях Python. Второй метод доступен для классов нового стиля в Python 2.X и для всех классов (неявно нового стиля) в Python 3.X. Эти два метода являются представителями набора методов перехвата атрибутов, куда также входят `__setattr__` и `__delattr__`. Тем не менее, так как данные методы исполняют похожие роли, мы будем трактовать их здесь в целом как одну тему.

В отличие от свойств и дескрипторов указанные методы относятся к универсальному протоколу *перегрузки операций* – множества особым образом именованных методов в классе, которые наследуются подклассами и автоматически запускаются, когда экземпляры используются в подразумеваемой встроенной операции. Подобно всем нормальным методам класса каждый из них при вызове принимает аргумент `self`, дающий доступ к любой необходимой информации состояния экземпляра, а также к другим методам класса, в котором они появляются.

Методы `__getattr__` и `__getattribute__` являются более *обобщенными*, чем свойства и дескрипторы – они могут применяться для перехвата операции извлечения любого атрибута экземпляра (или даже всех), а не только одного конкретного имени. В итоге эти два метода хорошо подходят в универсальных кодовых схемах, основанных на *делегировании* – они могут использоваться для реализации объектов оболочек (известных *как промежуточные* объекты), которые управляют всем доступом к внутреннему объекту. Наоборот, мы должны определять по одному свойству или дескриптору для каждого атрибута, который хотим перехватывать. Как вскоре будет показано, в классах нового стиля такая роль несколько неполноценна для встроенных операций, но по-прежнему применима ко всем именованным методам в интерфейсе внутреннего объекта.

Наконец, эти два метода более *узко ориентированы*, чем ранее рассмотренные альтернативы: они перехватывают только операции извлечения, но не присваивания. Чтобы перехватывать также изменения атрибутов посредством присваивания, нам

потребуется реализовать метод __setattr__ – метод перегрузки операций, запускаемый для присваивания каждого атрибута, который должен позаботиться об избегании рекурсивных циклов за счет прогона операций присваивания атрибутов через словарь пространств имен экземпляра или метод суперкласса. Хотя и менее часто, мы также можем реализовать метод перегрузки __delattr__ (который обязан избегать зацикливания аналогичным образом) для перехвата операций удаления атрибутов. Напротив, свойства и дескрипторы перехватывают операции извлечения, установки и *удаления* по определению.

Большинство методов перегрузки операций были представлены ранее в книге; здесь мы расширим их использование и выясним их роли в более крупных контекстах.

Основы

Методы __getattr__ и __setattr__ представлялись в главах 30 и 32, а метод __getattribute__ упоминался в главе 32. Говоря кратко, если класс определяет или наследует следующие методы, то они будут автоматически запускаться, когда экземпляр задействован в контексте, описанном справа в комментарии:

```
def __getattr__(self, name):  # При извлечении неопределенных атрибутов [obj.name]
def __getattribute__(self, name): # При извлечении всех атрибутов [obj.name]
def __setattr__(self, name, value): # При присваивании всех атрибутов [obj.name=value]
def __delattr__(self, name):      # При удалении всех атрибутов [del obj.name]
```

Во всех методах self представляет собой объект экземпляра, на котором они вызываются, name – строковое имя атрибута, подвергающегося доступу, и value – объект, присваиваемый атрибуту. Два метода извлечения обычно возвращают значение атрибута, другие два ничего не возвращают (None). Все они могут генерировать исключения, сигнализируя о запрете доступа.

Например, для перехвата извлечения каждого атрибута мы можем применять любой из первых двух описанных ранее методов, а для перехвата присваивания каждого атрибута – третий метод. Приведенный далее класс использует __getattr__ и *переносим* между Python 2.X и 3.X без необходимости в наследовании нового стиля от object в Python 2.X:

```
class Catcher:
    def __getattr__(self, name):
        print('Get: %s' % name)
    def __setattr__(self, name, value):
        print('Set: %s %s' % (name, value))

X = Catcher()
X.job                                   # Выводится Get: job
X.pay                                   # Выводится Get: pay
X.pay = 99                              # Выводится Set: pay 99
```

Применение __getattribute__ в этом конкретном случае работает точно так же, но требует (только) в Python 2.X наследования от object и обладает едва заметным потенциалом зацикливания, которым мы займемся в следующем разделе:

```
class Catcher(object):                  # В Python 2.X требуется (object)
    def __getattribute__(self, name):   # Работает здесь так же, как getattr
        print('Get: %s' % name)         # Но в целом подвержен зацикливанию
    ...остальной код остался прежним...
```

Такая кодовая структура может использоваться для реализации паттерна проектирования с *делегированием*, обсуждавшегося в главе 31. Поскольку доступ ко всем ат-

рибутам обобщенным образом направляется нашим методам перехвата, мы можем проверять и передавать их внедренным управляемым объектам. Скажем, в следующем классе (позаимствованном из главы 31) отслеживается извлечение *каждого* атрибута, сделанное другим объектом, который передается классу оболочки:

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Сохранить объект
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)              # Отслеживать извлечение
        return getattr(self.wrapped, attrname)   # Делегировать извлечение

X = Wrapper([1, 2, 3])
X.append(4)                                      # Выводится Trace: append
print(X.wrapped)                                 # Выводится [1, 2, 3, 4]
```

Такого аналога для свойств и дескрипторов не существует, если не считать реализацию методов доступа для *каждого* возможного атрибута в *каждом* объекте, который может быть внутренним. С другой стороны, когда такая универсальность не требуется, обобщенные методы доступа могут влечь за собой дополнительные вызовы для операций присваивания в ряде контекстов — компромисс, который описан в главе 30 и упоминается в учебном примере, рассматриваемом в конце главы.

Избегание циклов в методах перехвата атрибутов

В целом обсуждаемые здесь методы применять легко; единственный связанный с ними более сложный аспект — возможность *зацикливания* (т.е. рекурсии). Из-за того, что метод `__getattr__` вызывается только для неопределенных атрибутов, в своем коде он может свободно извлекать другие атрибуты. Однако поскольку `__getattribute__` и `__setattr__` запускаются для *всех* атрибутов, при доступе к другим атрибутам в их коде потребуется проявлять осмотрительность, чтобы избежать повторного вызова самих себя и образования рекурсивного цикла.

Например, извлечение еще одного атрибута внутри кода метода `__getattribute__` снова запустит `__getattribute__` и в коде обычно происходит зацикливание до тех пор, пока не будет исчерпана доступная память:

```
def __getattribute__(self, name):
    x = self.other                               # ЗАЦИКЛИВАНИЕ!
```

Формально метод `__getattribute__` еще более подвержен зацикливанию, чем можно было предполагать — ссылка на атрибут `self`, производимая *где угодно* в классе, который определяет этот метод, запустит `__getattribute__` и в зависимости от логики класса тоже обладает потенциалом зацикливания. Как правило, такое поведение является желательным — в конце концов, данный метод предназначен для перехвата извлечения каждого атрибута, — но вы должны осознавать, что метод перехватывает операции извлечения всех атрибутов, где бы они не находились в коде. При появлении в коде самого метода `__getattribute__` они почти всегда приводят к зацикливанию. Чтобы избежать зацикливания, взамен прогоняйте операцию извлечения через расположенный выше суперкласс, пропустив версию текущего уровня — так как класс `object` всегда выступает в качестве суперкласса нового стиля, он хорошо подходит для такой роли:

```
def __getattribute__(self, name):
    x = object.__getattribute__(self, 'other')   # Передача расположенному
                                                 #   выше суперклассу
```

Для метода __setattr__ ситуация аналогична, как было подытожено в главе 30 – присваивание *любому* атрибуту внутри данного метода приводит к повторному запуску __setattr__ и может создать похожий цикл:

```
def __setattr__(self, name, value):
    self.other = value                  # Рекурсия (и возможное ЗАЦИКЛИВАНИЕ!)
```

Здесь операции присваивания атрибуту self *в любом месте* класса, определяющего метод __setattr__, запускают этот метод снова, хотя потенциал зацикливания гораздо выше, когда операция присваивания атрибуту self находится в самом методе __setattr__. Чтобы обойти проблему, вы можете взамен выполнить присваивание атрибуту как ключу в словаре пространств имен __dict__ экземпляра, избежав прямого присваивания:

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value      # Использование словаря атрибутов
```

Есть и менее традиционный подход – во избежание зацикливания метод __setattr__ может также передавать собственные операции присваивания атрибутам расположенному выше суперклассу, почти как __getattribute__ (и согласно врезке "На заметку!" далее в главе такая схема временами предпочтительнее):

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value # Передача расположенному
                                            #   выше суперклассу
```

Тем не менее, для избегания циклов в __getattribute__ использовать прием с __dict__ *нельзя*:

```
def __getattribute__(self, name):
    x = self.__dict__['other']          # Зацикливание!
```

Извлечение самого атрибута __dict__ приводит к повторному запуску метода __getattribute__, образуя рекурсивный цикл. Странно, но это правда!

На практике метод __delattr__ применяется менее часто, но в случае реализации он вызывается для каждой операции удаления атрибута (в точности как __setattr__ вызывается для каждой операции присваивания атрибуту). Когда используется метод __delattr__, вы обязаны позаботиться об избегании зацикливания при удалении атрибутов посредством тех же самых методик: операций через словари пространств имен и обращений к методам суперкласса.

Как отмечалось в главе 30, атрибуты, которые реализованы с помощью средств классов нового стиля, таких как *слоты* и *свойства*, физически не хранятся в словаре пространств имен __dict__ экземпляра (и слоты могут даже воспрепятствовать его существованию). По указанной причине в коде, где желательно поддерживать атрибуты подобного рода, должен быть реализован метод __setattr__, чтобы выполнять присваивание по показанной выше схеме object.__setattr__, а не через индексирование self.__dict__. Операций с __dict__ достаточно для классов, о которых известно, что они хранят данные в экземплярах, как в автономных примерах, приводимых в главе; однако, в универсальных инструментах предпочтение должно отдаваться операциям с object.

Первый пример

Обобщенно управлять атрибутами не настолько сложно, как могло вытекать из предыдущего раздела. Давайте посмотрим, как изложенные идеи воплощаются на практике. Мы снова обращаемся к первому примеру, который применялся для демонстрации свойств и дескрипторов в действии, но на этот раз он реализован с использованием методов перегрузки операций. Из-за крайне обобщенного характера таких методов мы проверяем имена атрибутов, чтобы знать, когда происходит доступ к управляемому атрибуту; остальным атрибутам разрешено проходить нормально:

```
class Person:                           # Код переносимый: Python 2.X или 3.X
    def __init__(self, name):           # При [Person()]
        self._name = name               # Запускается __setattr__!
    def __getattr__(self, attr):        # При [obj.неопределенный_атрибут]
        print('get: ' + attr)
        if attr == 'name':      # Перехват имени name: не хранится в экземпляре
            return self._name           # Зацикливания нет: реальный атрибут
        else:                           # Остальные являются ошибками
            raise AttributeError(attr)
    def __setattr__(self, attr, value): # При [obj.любой_атрибут = value]
        print('set: ' + attr)
        if attr == 'name':
            attr = '_name'              # Установка внутреннего имени
        self.__dict__[attr] = value     # Избегание зацикливания
    def __delattr__(self, attr):        # При [del obj.любой_атрибут]
        print('del: ' + attr)
        if attr == 'name':
            attr = '_name'              # Избегание зацикливания,
        del self.__dict__[attr]         # но оно гораздо менее распространено
bob = Person('Bob Smith')           # Экземпляр bob имеет управляемый атрибут
print(bob.name)                         # Запускается __getattr__
bob.name = 'Robert Smith'               # Запускается __setattr__
print(bob.name)
del bob.name                            # Запускается __delattr__
print('-'*20)
sue = Person('Sue Jones')               # Экземпляр sue также наследует свойство
print(sue.name)
# print(Person.name.__doc__)            # Эквивалент здесь отсутствует
```

Обратите внимание, что присваивание атрибуту в конструкторе `__init__` тоже запускает метод `__setattr__` — он перехватывает операции присваивания *всем* атрибутам, даже внутри самого класса. При выполнении кода получается тот же самый вывод, но теперь он представляет собой результат работы нормального механизма перегрузки операций, поддерживаемого Python, и наших методов перехвата доступа к атрибутам:

```
c:\code> py -3 getattr-person.py
set: _name
get: name
Bob Smith
set: name
get: name
Robert Smith
del: name
--------------------
set: _name
get: name
Sue Jones
```

Также отметьте, что в отличие от свойств и дескрипторов здесь отсутствует прямое понятие указания *документации* для нашего атрибута; управляемые атрибуты существуют внутри кода методов перехвата, а не как отдельные объекты.

Использование __getattribute__

Для получения точно таких же результатов посредством __getattribute__ замените метод __getattr__ в примере приведенным далее кодом; поскольку он перехватывает извлечение *всех* атрибутов, в данной версии необходимо избегать зацикливания, передавая новые операции извлечения суперклассу, и в целом нельзя предполагать, что неизвестные имена являются ошибками:

```
# Замените __getattr__ следующим кодом

   def __getattribute__(self, attr):          # При [obj.любой_атрибут]
      print('get: ' + attr)
      if attr == 'name':                      # Перехват всех имен
         attr = '_name'                       # Отображение на внутреннее имя
      return object.__getattribute__(self, attr) # Избегание зацикливания
```

Запуск кода после внесения изменений дает похожий вывод, но мы имеем добавочный вызов __getattribute__ для операции извлечения в __setattr__ (в первый раз возникшей в __init__):

```
c:\code> py -3 getattribute-person.py
set: _name
get: __dict__
get: name
Bob Smith
set: name
get: __dict__
get: name
Robert Smith
del: name
get: __dict__
--------------------
set: _name
get: __dict__
get: name
Sue Jones
```

Пример эквивалентен тому, что был реализован для свойств и дескрипторов, но он несколько надуман и в действительности не подчеркивает возможности этих инструментов. Из-за своей обобщенной природы методы __getattr__ и __getattribute__ вероятно чаще применяются в коде, основанном на делегировании (как уже упоминалось), где операции доступа к атрибутам проверяются на предмет достоверности и направляются внутреннему объекту. Там, где необходимо управлять всего лишь *одним* атрибутом, свойства и дескрипторы могут подходить в равной степени хорошо или даже лучше.

Вычисляемые атрибуты

Как и ранее, наш предыдущий пример на самом деле не делал ничего кроме отслеживания операций извлечения атрибутов; вычисление значения атрибута при извлечении требует не намного больше работы. Что касается свойств и дескрипторов,

следующий код создает виртуальный атрибут X, при извлечении которого запускается вычисление:

```
class AttrSquare:
    def __init__(self, start):
        self.value = start              # Запускается __setattr__!

    def __getattr__(self, attr):        # При операциях извлечения
                                        #  неопределенных атрибутов
        if attr == 'X':
            return self.value ** 2      # value не является неопределенным
        else:
            raise AttributeError(attr)

    def __setattr__(self, attr, value):  # При операциях присваивания всех атрибутов
        if attr == 'X':
            attr = 'value'
        self.__dict__[attr] = value

A = AttrSquare(3)           # 2 экземпляра класса с перегрузкой
B = AttrSquare(32)          # Каждый имеет отличающуюся информацию состояния

print(A.X)                  # 3 ** 2
A.X = 4
print(A.X)                  # 4 ** 2
print(B.X)                  # 32 ** 2 (1024)
```

Результатом выполнения кода оказывается тот же самый вывод, который мы получали при использовании свойств и дескрипторов, но механика сценария основана на обобщенных методах перехвата атрибутов:

```
c:\code> py -3 getattr-computed.py
9
16
1024
```

Использование `__getattribute__`

Мы по-прежнему можем достичь того же эффекта с применением `__getattribute__` вместо `__getattr__`; ниже метод извлечения заменяется методом `__getattribute__`, а в метод установки `__setattr__` вносятся изменения, позволяющие избежать зацикливания за счет использования прямых обращений к методам суперкласса взамен присваивания по ключам `__dict__`:

```
class AttrSquare:                          # Добавить (object) для Python 2.X
    def __init__(self, start):
        self.value = start                 # Запускается __setattr__!

    def __getattribute__(self, attr): # При операциях извлечения всех атрибутов
        if attr == 'X':
            return self.value ** 2     # Снова запускается __getattribute__!
        else:
            return object.__getattribute__(self, attr)

    def __setattr__(self, attr, value): # При операциях присваивания всех атрибутов
        if attr == 'X':
            attr = 'value'
        object.__setattr__(self, attr, value)
```

Когда эта версия, находящаяся в файле getattribute-computed.py, запускается, то снова получаются те же самые результаты. Тем не менее, обратите внимание, что в методах класса происходит неявное перенаправление:

- self.value = start внутри конструктора запускает __setattr__;
- self.value внутри __getattribute__ снова запускает __getattribute__.

На самом деле метод __getattribute__ запускается *дважды* каждый раз, когда мы извлекаем атрибут X. В версии с __getattr__ подобное не происходит, потому что атрибут value не является неопределенным. Если вы заботитесь о скорости и хотите избежать двукратного вызова, тогда измените __getattribute__, чтобы для извлечения value также применять суперкласс:

```
def __getattribute__(self, attr):
    if attr == 'X':
        return object.__getattribute__(self, 'value') ** 2
```

Конечно, здесь по-прежнему происходит вызов метода суперкласса, но не дополнительный рекурсивный вызов до того, как мы туда доберемся. Добавление к методам вызовов print позволит выяснить, каким образом и когда они запускаются.

Сравнение __getattr__ и __getattribute__

Чтобы подытожить отличия между __getattr__ и __getattribute__, в следующем примере оба метода используются для реализации трех атрибутов — атрибута класса attr1, атрибута экземпляра attr2 и виртуального управляемого атрибута attr3, вычисляемого при извлечении:

```
class GetAttr:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattr__(self, attr):     # Только при операциях извлечения
                                     #   неопределенных атрибутов
        print('get: ' + attr)        # Не при извлечении атрибута attr1:
                                     #   наследуется из класса
        if attr == 'attr3':          # Не при извлечении атрибута attr2:
                                     #   хранится в экземпляре
            return 3
        else:
            raise AttributeError(attr)

X = GetAttr()
print(X.attr1)
print(X.attr2)
print(X.attr3)
print('-'*20)

class GetAttribute(object):           # Добавить (object) в Python 2.X
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattribute__(self, attr): # При операциях извлечения всех атрибутов
        print('get: ' + attr)         # Использование суперкласса во избежание
                                      #   зацикливания
        if attr == 'attr3':
            return 3
```

```
        else:
            return object.__getattribute__(self, attr)

X = GetAttribute()
print(X.attr1)
print(X.attr2)
print(X.attr3)
```

Версия с `__getattr__` перехватывает только доступ к атрибуту `attr3`, т.к. он не определен. С другой стороны, версия с `__getattribute__` перехватывает операции извлечения всех атрибутов и обязана направлять те, которыми она не управляет, методу извлечения из суперкласса, чтобы избежать появления циклов:

```
c:\code> py -3 getattr-v-getattr.py
1
2
get: attr3
3
--------------------
get: attr1
1
get: attr2
2
get: attr3
3
```

Хотя метод `__getattribute__` способен перехватывать больше операций извлечения атрибутов, чем `__getattr__`, на практике они часто являются лишь вариациями на тему – если атрибуты не хранятся физически, то оба метода обеспечивают тот же самый эффект.

Сравнение методик управления

Для подведения итогов по отличиям между всеми четырьмя схемами управления атрибутами, рассмотренными в главе, давайте проработаем более полный пример с вычисляемыми атрибутами, в котором применяется каждая методика и который рассчитан на выполнение в Python 3.X или 2.X. В показанной ниже первой версии с использованием *свойств* перехватываются и вычисляются атрибуты `square` и `cube`. Обратите внимание, что их базовые значения хранятся в именах, начинающихся с символа подчеркивания, чтобы они не конфликтовали с именами самих свойств:

```
# Два динамически вычисляемых атрибута, реализованные с помощью свойств

class Powers(object):                    # В Python 2.X требуется (object)
    def __init__(self, square, cube):
        self._square = square            # _square - базовое значение
        self._cube = cube                # square – имя свойства

    def getSquare(self):
        return self._square ** 2
    def setSquare(self, value):
        self._square = value
    square = property(getSquare, setSquare)

    def getCube(self):
        return self._cube ** 3
    cube = property(getCube)

X = Powers(3, 4)
```

```
print(X.square)          # 3 ** 2 = 9
print(X.cube)            # 4 ** 3 = 64
X.square = 5
print(X.square)          # 5 ** 2 = 25
```

Чтобы сделать то же самое посредством *дескрипторов*, мы определяем атрибуты с помощью полных классов. Обратите внимание, что дескрипторы хранят базовые значения в виде состояния экземпляра, поэтому они должны применять ведущие символы подчеркивания, чтобы не конфликтовать с именами дескрипторов. В финальном примере главы мы увидим, что такого требования по переименованию можно было бы избежать за счет хранения базовых значений как состояния дескрипторов, но это не касается непосредственно данных, которые должны варьироваться в зависимости от экземпляра клиентского класса:

```
# То же самое, но с помощью дескрипторов (состояние для каждого экземпляра)
class DescSquare(object):
    def __get__(self, instance, owner):
        return instance._square ** 2
    def __set__(self, instance, value):
        instance._square = value
class DescCube(object):
    def __get__(self, instance, owner):
        return instance._cube ** 3

class Powers(object):                     # В Python 2.X требуется (object)
    square = DescSquare()
    cube = DescCube()
    def __init__(self, square, cube):
        self._square = square             # self.square = square тоже работает,
        self._cube = cube        # т.к. приводит к запуску __set__ дескриптора!

X = Powers(3, 4)
print(X.square)          # 3 ** 2 = 9
print(X.cube)            # 4 ** 3 = 64
X.square = 5
print(X.square)          # 5 ** 2 = 25
```

Для получения того же результата с помощью метода перехвата извлечения `__getattr__` мы снова сохраняем базовые значения в именах, предваренных символами подчеркивания, так что управляемые имена при доступе оказываются неопределенными и потому вызывается наш метод. Нам также необходимо реализовать метод `__setattr__` для перехвата операций присваивания и позаботиться об устранении возможности зацикливания:

```
# То же самое, но с помощью обобщенного перехвата неопределенных
#   атрибутов методом __getattr__
class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube
    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        elif name == 'cube':
            return self._cube ** 3
        else:
            raise TypeError('unknown attr:' + name)
```

```
    def __setattr__(self, name, value):
        if name == 'square':
            self.__dict__['_square'] = value        # Или использовать object
        else:
            self.__dict__[name] = value

X = Powers(3, 4)
print(X.square)        # 3 ** 2 = 9
print(X.cube)          # 4 ** 3 = 64
X.square = 5
print(X.square)        # 5 ** 2 = 25
```

Последний вариант, в котором используется `__getattribute__`, похож на предыдущую версию. Однако поскольку теперь мы перехватываем доступ к каждому атрибуту, то должны также направлять операции извлечения базовых значений суперклассу, избегая зацикливания или добавочных вызовов — извлечение `self._square` напрямую тоже работает, но инициирует второй вызов `__getattribute__`:

```
# То же самое, но с помощью обобщенного перехвата неопределенных атрибутов
#   методом __getattribute__

class Powers(object):                        # В Python 2.X требуется (object)
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube

    def __getattribute__(self, name):
        if name == 'square':
            return object.__getattribute__(self, '_square') ** 2
        elif name == 'cube':
            return object.__getattribute__(self, '_cube') ** 3
        else:
            return object.__getattribute__(self, name)

    def __setattr__(self, name, value):
    if name == 'square':
      object.__setattr__(self, '_square', value) # Либо использовать __dict__
    else:
        object.__setattr__(self, name , value)

X = Powers(3, 4)
print(X.square)        # 3 ** 2 = 9
print(X.cube)          # 4 ** 3 = 64
X.square = 5
print(X.square)        # 5 ** 2 = 25
```

Несмотря на отличающиеся формы, которые каждая методика принимает в коде, все четыре при выполнении производят одинаковые результаты:

```
9
64
25
```

Дополнительные сведения, касающиеся сравнения таких альтернатив, и другие варианты реализации будут предоставлены при разработке более реалистичного приложения в разделе “Пример: проверка достоверности атрибутов” далее в главе. Тем не менее, сначала нам нужно кратко ознакомиться со связанной с классами нового стиля ловушкой, которая скрыта в двух этих инструментах — обобщенными методами перехвата, описанными в текущем разделе.

Перехват атрибутов для встроенных операций

Если вы читали настоящую книгу последовательно, тогда часть данного раздела будет обзором и уточнением материалов, раскрытых ранее, особенно в главе 32. Для остальных тема представлена в главе в надлежащем контексте.

Во время представления методов `__getattr__` и `__getattribute__` я заявил о том, что они перехватывают операции извлечения соответственно неопределенных и всех атрибутов, идеально подходя для кодовых схем с делегированием. Хотя сказанное справедливо в отношении *нормально именованных* и *явно вызываемых* атрибутов, их поведение требует дополнительного прояснения: для атрибутов имен методов, неявно извлекаемых *встроенными* операциями, такие методы могут *вообще не запускаться*. Это означает, что вызовы методов перегрузки операций не могут делегироваться внутренним объектам, если только классы оболочек самостоятельно не переопределят данные методы.

Скажем, операции извлечения атрибутов для методов `__str__`, `__add__` и `__getitem__`, запускаемые неявно соответственно выводом, выражениями + и индексированием, в Python 3.X не направляются методам перехвата атрибутов. В частности:

- в Python 3.X ни `getattr__`, ни `__getattribute__` не запускаются для таких атрибутов;
- в классических классах Python 2.X метод `__getattr__` запускается для таких атрибутов, если они определены в классе;
- в Python 2.X метод `__getattribute__` доступен только для классов нового стиля и работает так, как в Python 3.X.

Другими словами, во всех классах Python 3.X (и классах нового стиля Python 2.X) не существует прямого способа обобщенного перехвата встроенных операций вроде вывода и сложения. В стандартных классических классах Python 2.X методы таких операций ищутся во время выполнения в *экземплярах* подобно всем остальным атрибутам; в классах нового стиля Python 3.X такие методы взамен ищутся в *классах*. Поскольку в Python 3.X классы нового стиля обязательны, а в Python 2.X по умолчанию применяются классические классы, это вполне естественно для Python 3.X, но может также произойти в коде нового стиля Python 2.X. Однако в Python 2.X вы, по крайней мере, располагаете способом избежать такого изменения, тогда как в Python 3.X – нет.

Согласно главе 32 официальное (хотя и скудно документированное) обоснование для данного изменения, похоже, вращается вокруг метаклассов и оптимизации встроенных операций. С учетом того, что все атрибуты – нормально именованные и другие – при *явном* доступе по имени по-прежнему направляются обобщенным образом через экземпляр и упомянутые методы, как представляется, это не препятствует делегированию в целом; оно больше похоже на шаг для оптимизации неявного поведения встроенных операций. Тем не менее, в итоге кодовые схемы, основанные на делегировании, в Python 3.X становятся более сложными, т.к. промежуточные классы для объектных интерфейсов не могут обобщенно перехватывать вызовы методов перегрузки операций и направлять их внутренним объектам.

Это неудобство, но не обязательно непреодолимое препятствие – классы оболочек могут обойти ограничение, самостоятельно переопределяя все имеющие отношение к делу методы перегрузки операций, чтобы делегировать вызовы. Дополнительные методы можно добавить вручную, посредством инструментов или путем их определения и наследования от общих суперклассов. Однако в результате объекты-оболочки

требуют больше работы, чем обычно, когда методы перегрузки операций являются частью интерфейса внутреннего объекта.

Имейте в виду, что проблема касается только методов `__getattr__` и `__getattribute__`. Поскольку свойства и дескрипторы определяются только для конкретных атрибутов, в действительности они вообще не применяются к классам, основанным на делегировании — единственное свойство или дескриптор не может использоваться для перехвата произвольных атрибутов. Более того, класс, в котором определены *и* методы перегрузки операций, *и* перехват атрибутов, будут работать корректно безотносительно к типу определенного перехвата атрибутов. Здесь мы заботимся только о классах, которые не имеют определенных методов перегрузки операций, но пытаются перехватывать их обобщенным образом.

Рассмотрим следующий пример из файла `getattr-bultins.py`, где тестируются различные типы атрибутов и встроенных операций на экземплярах класса, содержащего методы `__getattr__` и `__getattribute__`:

```
class GetAttr:
    eggs = 88                      # eggs хранится в классе, spam - в экземпляре
    def __init__(self):
        self.spam = 77
    def __len__(self):                 # Реализовать здесь len,
                                       #   иначе __getattr__ вызывается с __len__
        print('__len__: 42')
        return 42
    def __getattr__(self, attr):       # Предоставить __str__ по запросу,
                                       #   иначе фиктивную функцию
        print('getattr: ' + attr)
        if attr == '__str__':
            return lambda *args: '[Getattr str]'
        else:
            return lambda *args: None

class GetAttribute(object):            # object требуется в Python 2.X
                                       #   и подразумевается в Python 3.X
    eggs = 88                          # В Python 2.X все автоматически
                                       #   isinstance(object)
    def __init__(self):                # Но нужно наследовать от object,
                                       #   чтобы получить инструменты нового
        self.spam = 77                 # стиля, включая __getattribute__
                                       #   и ряд стандартных методов __X__
    def __len__(self):
        print('__len__: 42')
        return 42
    def __getattribute__(self, attr):
        print('getattribute: ' + attr)
        if attr == '__str__':
            return lambda *args: '[GetAttribute str]'
        else:
            return lambda *args: None
for Class in GetAttr, GetAttribute:
    print('\n' + Class.__name__.ljust(50, '='))
    X = Class()
    X.eggs                  # Атрибут класса
    X.spam                  # Атрибут экземпляра
    X.other                 # Отсутствующий атрибут
    len(X)                  # __len__ определено явно
```

```
# Классы нового стиля обязаны поддерживать [], +, прямой вызов: переопределить
try: X[0]              # __getitem__?
except: print('fail []')

try: X + 99            # __add__?
except: print('fail +')

try: X()               # __call__? (неявно через встроенную операцию)
except: print('fail ()')

X.__call__()           # __call__? (явно, не наследуется)
print(X.__str__())     # __str__? (явно, наследуется от типа)
print(X)               # __str__? (неявно через встроенную операцию)
```

При запуске в том виде, как есть, под управлением Python 2.X метод `__getattr__` *будет* получать разнообразные запросы на неявное извлечение атрибутов для встроенных операций, потому что при нормальных обстоятельствах интерпретатор Python ищет такие атрибуты в экземплярах. И наоборот, `__getattribute__` *не будет* запускаться для любых имен перегрузки операций, вызываемых встроенными операциями, поскольку в модели классов нового стиля такие имена ищутся только в классах:

```
c:\code> py -2 getattr-builtins.py
GetAttr==================================================
getattr: other
__len__: 42
getattr: __getitem__
getattr: __coerce__
getattr: __add__
getattr: __call__
getattr: __call__
getattr: __str__
[Getattr str]
getattr: __str__
[Getattr str]
GetAttribute=============================================
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
fail []
fail +
fail ()
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x02287898>
```

Обратите внимание на то, как метод `__getattr__` перехватывает неявные и явные извлечения `__call__` и `__str__` в Python 2.X. По контрасту с этим метод `__getattribute__` отказывается перехватывать неявные извлечения любого из двух имен атрибутов для встроенных операций.

На самом деле ситуация с `__getattribute__` в Python 2.X будет такой же, как в Python 3.X, потому что для применения данного метода в Python 2.X классы должны быть превращены в классы нового стиля за счет их наследования от `object`. Наследовать от `object` в Python 3.X необязательно, т.к. там все классы являются классами нового стиля.

Тем не менее, при запуске под управлением Python 3.X результаты для `__getattr__` отличаются — *ни один* из неявно выполняемых методов перегрузки операций не запускает *тот или другой* метод перехвата атрибутов, когда их атрибуты извлекаются встроенными операциями. При распознавании таких имен интерпретатор Python 3.X (и классы нового стиля в целом) обходят обычный механизм поиска в экземпляре, хотя нормально именованные методы по-прежнему перехватываются, как и ранее:

```
c:\code> py -3 getattr-builtins.py
GetAttr===========================================
getattr: other
__len__: 42
fail []
fail +
fail ()
getattr: __call__
<__main__.GetAttr object at 0x02987CC0>
<__main__.GetAttr object at 0x02987CC0>
GetAttribute======================================
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
fail []
fail +
fail ()
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x02987CF8>
```

Имея вывод, отыщите соответствующие вызовы `print` в сценарии, чтобы понять его работу. Ниже отмечено несколько важных моментов.

- Доступ к методу `__str__` не удалось перехватить дважды методом `__getattr__` в Python 3.X: один раз для встроенной операции вывода и один раз для явных извлечений из-за наследования стандартного метода от класса (на самом деле от встроенного класса `object`, который является автоматическим суперклассом для каждого класса в Python 3.X).
- Доступ к методу `__str__` не удалось перехватить только раз универсальным обработчиком `__getattribute__` во время выполнения встроенной операции вывода; явные извлечения обходят унаследованную версию.
- Доступ к методу `__call__` не удалось перехватить в обеих схемах Python 3.X для выражений вызова встроенных операций, но он перехватывается обеими схемами при явном извлечении; в отличие от `__str__` для экземпляров `object` не существует унаследованного стандартного метода `__call__`, чтобы привести к неудаче `__getattr__`.
- Доступ к методу `__len__` перехватывается обоими классами просто потому, что он является явно определенным методом в самих классах — хотя в Python 3.X его имя не направляется к `__getattr__` или `__getattribute__`, если мы удаляем методы `__len__` класса.
- Все остальные встроенные операции не удалось перехватить обеими схемами в Python 3.X.

И снова совокупный эффект заключается в том, что методы перегрузки операций, неявно запускаемые встроенными операциями, никогда не прогоняются через любой из двух методов перехвата атрибутов в Python 3.X: классы нового стиля Python 3.X ищут такие атрибуты в классах, полностью пропуская шаг поиска в экземплярах. Нормально именованных атрибутов это не касается.

Такая особенность делает классы оболочек, основанные на делегировании, более сложными в реализации с помощью классов нового стиля Python 3.X – если внутренние классы могут содержать методы перегрузки операций, то эти методы придется избыточно переопределять в классе оболочки, чтобы делегировать работу внутреннему объекту. В универсальных инструментах делегирования может потребоваться добавить десятки дополнительных методов.

Разумеется, добавление таких методов можно отчасти автоматизировать посредством инструментов, дополняющих классы новыми методами (здесь способны помочь декораторы классов и метаклассы, рассматриваемые в последующих двух главах). Кроме того, суперкласс может быть в состоянии один раз определить все дополнительные методы для наследования в классах, основанных на делегировании. И все же кодовые схемы делегирования в классах Python 3.X требуют выполнения добавочной работы.

Более реалистичная иллюстрация данного явления вместе с обходным приемом представлена в примере декоратора `Private` в следующей главе. Там мы исследуем альтернативы реализации методов операций, требуемых промежуточными классами в Python 3.X, включая модели с многократно используемыми *подмешиваемыми суперклассами*. Также будет показано, что в клиентский класс можно вставить метод `__getattribute__`, чтобы предохранить его исходный тип, хотя данный метод по-прежнему не будет вызываться для методов перегрузки операций; например, вывод все еще запускает метод `__str__`, непосредственно определенный в таком классе, а не прогоняет запрос через `__getattribute__`.

В качестве реальной демонстрации в следующем разделе возрождается наш учебный пример по классам. Теперь, когда вы понимаете, как работает перехват атрибутов, у меня появилась возможность объяснить один из странных моментов, связанных с ним.

Снова о классах для регистрации и обработки сведений о людях, основанных на делегировании

В обучающем руководстве по ООП в главе 28 был представлен класс `Manager`, где применялось внедрение объектов и делегирование выполнения методов для настройки его суперкласса вместо наследования. Ниже приведен его код, из которого удалено не относящееся к делу тестирование:

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)
```

```
class Manager:
   def __init__(self, name, pay):
      self.person = Person(name, 'mgr', pay)     # Внедрение объекта Person
   def giveRaise(self, percent, bonus=.10):
      self.person.giveRaise(percent + bonus)     # Перехват и делегирование
   def __getattr__(self, attr):
      return getattr(self.person, attr)          # Делегирование всех
                                                 #   остальных атрибутов
   def __repr__(self):
      return str(self.person)    # Снова требуется перегрузка (в Python 3.X)
if __name__ == '__main__':
   sue = Person('Sue Jones', job='dev', pay=100000)
   print(sue.lastName())
   sue.giveRaise(.10)
   print(sue)
   tom = Manager('Tom Jones', 50000)             # Manager.__init__
   print(tom.lastName())       # Manager.__getattr__ -> Person.lastName
   tom.giveRaise(.10)          # Manager.giveRaise -> Person.giveRaise
   print(tom)                  # Manager.__repr__ -> Person.__repr__
```

Комментарии в конце файла показывают, какие методы вызываются для операции в каждой строке. В частности, обратите внимание на то, что вызовы lastName не определены в классе Manager и потому направляются обобщенному методу __getattr__, а оттуда внутреннему объекту Person. Далее приведен вывод сценария — объект sue получает повышение на 10% от Person, но объект tom — на 20%, т.к. метод giveRaise был настроен в классе Manager:

```
c:\code> py -3 getattr-delegate.py
Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

Однако по контрасту с этим взгляните, что происходит, когда мы *выводим* объект Manager в конце сценария: вызывается метод __repr__ класса оболочки и делегирует выполнение работы методу __repr__ внедренного объекта Person. Имея данный факт в виду, посмотрим, что случится, если мы *удалим* метод Manager.__repr__:

```
# Удаление метода __str__ в классе Manager

class Manager:
   def __init__(self, name, pay):
      self.person = Person(name, 'mgr', pay)     # Внедрение объекта Person
   def giveRaise(self, percent, bonus=.10):
      self.person.giveRaise(percent + bonus)     # Перехват и делегирование
   def __getattr__(self, attr):
      return getattr(self.person, attr)          # Делегирование всех
                                                 #   остальных атрибутов
```

Теперь в случае классов нового стиля Python 3.X вывод *не* прогоняет операции извлечения своих атрибутов через обобщенный метод перехвата __getattr__ для объектов Manager. Взамен находится и запускается стандартный метод отображения __repr__, унаследованный из неявного суперкласса object для класса (объект sue по-прежнему выводится корректно, потому что класс Person имеет явный метод __repr__):

```
c:\code> py -3 getattr-delegate.py
Jones
[Person: Sue Jones, 110000]
Jones
<__main__.Manager object at 0x029E7B70>
```

Выполнение без метода `__repr__` *запускает* метод `__getattr__` в классических классах Python 2.X, т.к. атрибуты перегрузки операций прогоняются через данный метод и такие классы не наследуют стандартный метод `__repr__`:

```
c:\code> py -2 getattr-delegate.py
Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

Переключение на метод `__getattribute__` здесь не поможет интерпретатору Python 3.X – подобно `__getattr__` он не запускается для атрибутов перегрузки операций, подразумеваемых встроенными операциями в Python 2.X или в Python 3.X:

```
# Замена __getattr_ методом __getattribute__

class Manager(object):                       # Использовать (object) в Python 2.X
   def __init__(self, name, pay):
      self.person = Person(name, 'mgr', pay)     # Внедрение объекта Person
   def giveRaise(self, percent, bonus=.10):
      self.person.giveRaise(percent + bonus)     # Перехват и делегирование
   def __getattribute__(self, attr):
     print('**', attr)
     if attr in ['person', 'giveRaise']:
       return object.__getattribute__(self, attr) # Извлечение моих атрибутов
     else:
       return getattr(self.person, attr)  # Делегирование всех остальных атрибутов
```

Независимо от того, какой метод перехвата атрибутов используется в Python 3.X, мы все равно должны включать в `Manager` переопределенный метод `__repr__` (как было показано ранее), чтобы перехватывать операции вывода и направлять их внедренному объекту `Person`:

```
C:\code> py -3 getattr-delegate.py
Jones
[Person: Sue Jones, 110000]
** lastName
** person
Jones
** giveRaise
** person
<__main__.Manager object at 0x028E0590>
```

Обратите внимание на то, что `__getattribute__` вызывается для методов *дважды* – один раз для имени метода и еще раз для извлечения внедренного объекта `self.person`. Этого можно было бы избежать за счет написания отличающегося кода, но мы по-прежнему обязаны переопределять метод `__repr__` для перехвата вывода, хотя и по-другому (`self.person` мог бы привести к отказу `__getattribute__`):

```
# Другая реализация __getattribute__ с целью минимизации добавочных вызовов
class Manager:
   def __init__(self, name, pay):
      self.person = Person(name, 'mgr', pay)
```

```
    def __getattribute__(self, attr):
        print('**', attr)
        person = object.__getattribute__(self, 'person')
        if attr == 'giveRaise':
            return lambda percent: person.giveRaise(percent+.10)
        else:
            return getattr(person, attr)
    def __repr__(self):
        person = object.__getattribute__(self, 'person')
        return str(person)
```

При выполнении такой альтернативной версии наш объект выводится надлежащим образом, но лишь потому, что мы добавили в класс оболочки явный метод `__repr__` – данный атрибут не направляется имеющемуся обобщенному методу перехвата атрибутов:

```
Jones
[Person: Sue Jones, 110000]
** lastName
Jones
** giveRaise
[Person: Tom Jones, 60000]
```

Вкратце история заключается в том, что основанные на делегировании классы вроде `Manager` должны переопределять некоторые методы перегрузки операций (подобные `__repr__` и `__str__`) для их направления внедренным объектам в Python 3.X, но не в Python 2.X, если только не применяются классы нового стиля. Похоже, нам доступны лишь варианты использования `__getattr__` и Python 2.X либо избыточного переопределения методов перегрузки операций для классов оболочек в Python 3.X.

Опять-таки задача не считается невыполнимой; многие классы оболочек могут спрогнозировать требуемый набор методов перегрузки операций, а инструменты и суперклассы способны автоматизировать часть решения задачи – на самом деле мы изучим необходимые кодовые схемы в следующей главе. Кроме того, не все классы используют методы перегрузки операций (в действительности большинство классов приложений обычно не должны их применять). Тем не менее, об этом важно помнить при работе с моделями делегирования в Python 3.X; когда методы перегрузки операций являются частью интерфейса объекта, то классы оболочек обязаны приспособиться к ним переносимым образом за счет локального переопределения.

Пример: проверка достоверности атрибутов

В заключение главы давайте рассмотрим более реалистичный пример, реализующий все четыре схемы управления атрибутами. В примере будет определен объект `CardHolder` с четырьмя атрибутами, три из которых управляемы. При излечении и сохранении управляемые атрибуты подвергаются проверке достоверности или видоизменению. Для того же самого тестового кода все четыре версии производят одинаковые результаты, но они реализуют свои атрибуты совершенно по-разному. Примеры предназначены в основном для самостоятельного изучения; хотя я не буду подробно останавливаться на их коде, все они используют концепции, уже исследованные в данной главе.

Использование свойств для проверки достоверности

В первой версии для управления тремя атрибутами применяются свойства. Как обычно, вместо управляемых атрибутов мы могли бы использовать простые методы, но свойства помогут, если атрибуты уже были задействованы в существующем коде. Свойства запускают код автоматически при доступе к атрибутам, но сосредоточены на конкретном наборе атрибутов; они не могут применяться для перехвата всех атрибутов обобщенным образом.

Чтобы понять этот код, важно помнить о том, что операции присваивания атрибутов внутри метода конструктора `__init__` тоже запускают методы установки свойств. Например, когда в `__init__` присваивается значение `self.name`, автоматически вызывается метод `setName`, который видоизменяет значение и присваивает его атрибуту экземпляра по имени `__name`, не конфликтующему с именем самого свойства.

Такое переименование (иногда называемое *корректировкой имен*) является необходимым, поскольку свойства используют общее состояние экземпляра, не имея собственного. Данные хранятся в атрибуте по имени `__name`, тогда как атрибут по имени `name` – всегда свойство, а не данные. В главе 31 мы выяснили, что имена вроде `__name` известны как *псевдозакрытые* атрибуты, которые при сохранении в пространстве имен экземпляра интерпретатор Python изменяет, добавляя имя включающего класса; здесь это помогает отличать атрибуты, специфичные к реализации, от остальных, в том числе от управляющего ими свойства.

В конечном счете, реализованный класс управляет атрибутами `name`, `age` и `acct`, разрешает прямой доступ к атрибуту `addr` и предоставляет атрибут только для чтения по имени `remain`, который является полностью виртуальным и вычисляется по запросу. Для сравнения версия на основе свойств занимает 39 строк кода, не считая двух начальных строк, и включает наследование от `object`, требующееся в Python 2.X, но необязательное в Python 3.X:

```
# Файл validate_properties.py

class CardHolder(object):                    # В Python 2.X требуется (object)
    acctlen = 8                              # Данные класса
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                     # Данные экземпляра
        self.name = name                     # Тоже запускают методы установки
                                             #   свойств!
        self.age = age                       # Имя __X корректируется, чтобы
                                             #   содержать имя класса
        self.addr = addr                     # Имя addr не корректируется
                                             # Свойство remain не имеет данных

    def getName(self):
        return self.__name
    def setName(self, value):
        value = value.lower().replace(' ', '_')
        self.__name = value
    name = property(getName, setName)

    def getAge(self):
        return self.__age
    def setAge(self, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age'    # недопустимый возраст
```

```
        else:
            self.__age = value
    age = property(getAge, setAge)

    def getAcct(self):
        return self.__acct[:-3] + '***'
    def setAcct(self, value):
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('invald acct number') # недопустимый номер счета
        else:
            self.__acct = value
    acct = property(getAcct, setAcct)

    def remainGet(self):                    # Могло быть методом, а не атрибутом,
        return self.retireage - self.age    #   если только уже не используется
                                            #   как атрибут
    remain = property(remainGet)
```

Тестовый код

Показанный далее код из файла `validate_tester.py` тестирует наш класс; запускайте этот сценарий, передавая имя модуля класса (без `.py`) как единственный аргумент командной строки (большую часть тестового кода можно было бы также добавить в конец каждого файла либо интерактивно импортировать его из модуля после импортирования класса). Для тестирования всех четырех версий в примере мы будем применять тот же самый код. После запуска он создает два экземпляра нашего класса с управляемыми атрибутами, после чего извлекает и изменяет различные атрибуты. Операции с ожидаемым отказом помещены внутрь операторов `try`, а идентичное поведение в Python 2.X поддерживается за счет включения функции `print` из Python 3.X:

```
# Файл validate_tester.py
from __future__ import print_function           # Python 2.X

def loadclass():
    import sys, importlib
    modulename = sys.argv[1]                     # Имя модуля в командной строке
    module = importlib.import_module(modulename) # Импортирование модуля
                                                 #   по имени в строке
    print('[Using: %s]' % module.CardHolder)     # getattr() здесь не требуется
    return module.CardHolder

def printholder(who):
    print(who.acct, who.name, who.age, who.remain, who.addr, sep=' / ')

if __name__ == '__main__':
    CardHolder = loadclass()
    bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
    printholder(bob)
    bob.name = 'Bob Q. Smith'
    bob.age = 50
    bob.acct = '23-45-67-89'
    printholder(bob)

    sue = CardHolder('5678-12-34', 'Sue Jones', 35, '124 main st')
    printholder(sue)
    try:
        sue.age = 200
```

```
    except:
        print('Bad age for Sue')          # Недопустимый возраст для sue
    try:
        sue.remain = 5
    except:
        print("Can't set sue.remain")     # Невозможно установить sue.remain
    try:
        sue.acct = '1234567'
    except:
        print('Bad acct for Sue')         # Недопустимый номер счета для sue
```

Ниже приведен вывод кода самотестирования в Python 3.X и 2.X; он одинаков для всех четырех версий примера кроме имени тестируемого класса. Отследите код, чтобы посмотреть, как вызываются методы класса; номера расчетных счетов отображаются с несколькими скрытыми цифрами, имена преобразуются в стандартный формат, а время, оставшееся до выхода на пенсию, вычисляется при извлечении с использованием вычитания атрибутов класса:

```
c:\code> py -3 validate_tester.py validate_properties
[Using: <class 'validate_properties.CardHolder'>]
12345*** / bob_smith / 40 / 19.5 / 123 main st
23456*** / bob_q._smith / 50 / 9.5 / 123 main st
56781*** / sue_jones / 35 / 24.5 / 124 main st
Bad age for Sue
Can't set sue.remain
Bad acct for Sue
```

Использование дескрипторов для проверки достоверности

А теперь давайте перепишем наш пример с применением *дескрипторов* вместо свойств. Как демонстрировалось ранее, дескрипторы очень похожи на свойства в плане функциональности и ролей; на самом деле свойства по существу представляют собой ограниченную форму дескрипторов. Подобно свойствам дескрипторы предназначены для обработки конкретных атрибутов, а не обобщенного доступа к атрибутам. В отличие от свойств дескрипторы могут также иметь собственное состояние и являются более универсальной схемой.

Вариант 1: проверка достоверности с помощью разделяемого состояния экземпляра дескриптора

Для понимания показанного далее кода снова важно помнить о том, что операции присваивания значений атрибутам внутри метода конструктора `__init__` запускают методы `__set__` дескриптора. Скажем, когда в методе конструктора выполняется присваивание `self.name`, то автоматически вызывается метод `Name.__set__()`, который видоизменяет значение и присваивает его атрибуту дескриптора по имени `name`.

В конце концов, класс реализует те же самые атрибуты, что и предыдущая версия: он управляет атрибутами `name`, `age` и `acct`, разрешает прямой доступ к атрибуту `addr` и предоставляет атрибут только для чтения по имени `remain`, который является полностью виртуальным и вычисляется по запросу. Обратите внимание, что мы обязаны перехватывать операции присваивания значений имени `remain` в его дескрипторе и генерировать исключение; как объяснялось ранее, если не поступать так, тогда операция присваивания этому атрибуту молча создаст атрибут экземпляра, который скроет дескриптор атрибута класса.

Для сравнения версия на основе дескрипторов требует 45 строк кода; я добавил обязательное наследование от object к основным классам дескрипторов ради совместимости с Python 2.X (его можно опустить в коде, запускаемом только в Python 3.X, но оно ничем не вредит и содействует переносимости):

```
# Файл validate_descriptors1.py: использование разделяемого состояния
#   экземпляра дескриптора

class CardHolder(object):                  # В Python 2.X требуется (object)
    acctlen = 8                            # Данные класса
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                   # Данные экземпляра
        self.name = name                   # Тоже запускают методы __set__!
        self.age = age                     # Имя __X не требуется: в дескрипторе
        self.addr = addr                   # Имя addr не является управляемым
                                           # remain не имеет данных
    class Name(object):
        def __get__(self, instance, owner): # Имена классов: локальные в CardHolder
            return self.name
        def __set__(self, instance, value):
            value = value.lower().replace(' ', '_')
            self.name = value
    name = Name()

    class Age(object):
        def __get__(self, instance, owner):
            return self.age                # Использовать данные дескриптора
        def __set__(self, instance, value):
            if value < 0 or value > 150:
                raise ValueError('invalid age')  # недопустимый возраст
            else:
                self.age = value
    age = Age()

    class Acct(object):
        def __get__(self, instance, owner):
            return self.acct[:-3] + '***'
        def __set__(self, instance, value):
            value = value.replace('-', '')
            if len(value) != instance.acctlen:      # Использовать данные
                                                    #   экземпляра класса
                raise TypeError('invald acct number') # недопустимый номер счета
            else:
                self.acct = value
    acct = Acct()

    class Remain(object):
        def __get__(self, instance, owner):
            return instance.retireage - instance.age # Запускается Age.__get__
        def __set__(self, instance, value):
            raise TypeError('cannot set remain')     # Установка не разрешена
    remain = Remain()
```

При запуске с предыдущим тестовым сценарием все примеры в настоящем разделе производят тот же самый вывод, показанный ранее для версии со свойствами, за исключением отличающегося имени класса в первой строке:

```
C:\code> python validate_tester.py validate_descriptors1
...тот же самый вывод, что и в версии со свойствами, кроме имени класса...
```

Вариант 2: проверка достоверности с помощью состояния для каждого экземпляра клиентского класса

Однако в отличие от предшествующего варианта, основанного на свойствах, в этом случае действительное значение name присоединяется к объекту *дескриптора*, а не к экземпляру клиентского класса. Хотя мы могли бы хранить значение name либо в состоянии экземпляра, либо в состоянии дескриптора, в последней ситуации устраняется необходимость в корректировке имен за счет добавления символов подчеркивания во избежание конфликтов. В клиентском классе CardHolder атрибут по имени name всегда будет не данными, а объектом дескриптора.

Важно упомянуть о недостатке такой схемы — состояние, хранящееся внутри самого дескриптора, представляет собой данные уровня класса, которые фактически *разделяются* всеми экземплярами клиентского класса и потому не могут варьироваться между ними. То есть хранение состояния в экземпляре *дескриптора* вместо экземпляра *владеющего* (клиентского) класса означает, что состояние будет тем же самым во всех экземплярах владеющего класса. Состояние дескриптора может отличаться только для каждого атрибута.

Чтобы увидеть подход в работе, мы попробуем в предыдущей версии класса CardHolder, основанной на дескрипторах, вывести атрибуты экземпляра bob после создания второго экземпляра, sue. Значения управляемых атрибутов в sue (name, age и acct) *переписывают* значения одноименных атрибутов в ранее созданном объекте bob, потому что оба объекта разделяют тот же самый одиночный экземпляр дескриптора, присоединенный к классу:

```
# Файл validate_tester2.py
from __future__ import print_function                    # Python 2.X

from validate_tester import loadclass
CardHolder = loadclass()

bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
print('bob:', bob.name, bob.acct, bob.age, bob.addr)

sue = CardHolder('5678-12-34', 'Sue Jones', 35, '124 main st')
print('sue:', sue.name, sue.acct, sue.age, sue.addr) # addr отличается:
                                                     #   клиентские данные
print('bob:', bob.name, bob.acct, bob.age, bob.addr)  # name, acct, age
                                                      #   переписываются?
```

Результаты подтверждают подозрение — с точки зрения управляемых атрибутов объект bob превратился в sue!

```
c:\code> py -3 validate_tester2.py validate_descriptors1
[Using: <class 'validate_descriptors1.CardHolder'>]
bob: bob_smith 12345*** 40 123 main st
sue: sue_jones 56781*** 35 124 main st
bob: sue_jones 56781*** 35 123 main st
```

Разумеется, существуют допустимые сценарии использования для состояния дескриптора — управление реализацией дескриптора и данными, охватывающими все экземпляры, — и код был реализован в целях иллюстрации методики. Кроме того, в этом месте книги последствия, касающиеся области видимости состояния, для атрибутов класса и экземпляров должны быть более-менее ясны.

Тем не менее, в рассматриваемом конкретном сценарии атрибуты объектов CardHolder вероятно лучше хранить в виде данных *для каждого экземпляра*, а не данных экземпляра дескриптора, возможно с применением того же соглашения об именовании __*X*, которое использовалось в версии на основе свойств, чтобы избежать конфликтов имен в экземпляре — на этот раз более важный фактор, т.к. клиентом является другой класс с собственными атрибутами состояния. Вот необходимые изменения в коде; количество строк осталось прежним (45):

```
# Файл validate_descriptors2.py: использование состояния для каждого
экземпляра клиентского класса

class CardHolder(object):            # В Python 2.X требуется (object)
    acctlen = 8                      # Данные класса
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct             # Данные экземпляра клиентского класса
        self.name = name             # Тоже запускают методы __set__!
        self.age = age               # Имя __X требуется: в экземпляре клиента
        self.addr = addr             # Имя addr не является управляемым
                                     # remain является управляемым,
                                     #   но не имеет данных
    class Name(object):
        def __get__(self, instance, owner):  # Имена классов: локальные в CardHolder
            return instance.__name
        def __set__(self, instance, value):
            value = value.lower().replace(' ', '_')
            instance.__name = value
    name = Name()                # class.name или скорректированное имя атрибута

    class Age(object):
        def __get__(self, instance, owner):
            return instance.__age                # Использовать данные дескриптора
        def __set__(self, instance, value):
            if value < 0 or value > 150:
                raise ValueError('invalid age') # недопустимый возраст
            else:
                instance.__age = value
    age = Age()                  # class.age или скорректированное имя атрибута

    class Acct(object):
        def __get__(self, instance, owner):
            return instance.__acct[:-3] + '***'
        def __set__(self, instance, value):
            value = value.replace('-', '')
            if len(value) != instance.acctlen:       # Использовать данные
                                                     #   экземпляра класса
                raise TypeError('invald acct number') # недопустимый номер счета
            else:
                instance.__acct = value
    acct = Acct()                # class.acct или скорректированное имя атрибута

    class Remain(object):
        def __get__(self, instance, owner):
            return instance.retireage - instance.age # Запускается Age.__get__
        def __set__(self, instance, value):
            raise TypeError('cannot set remain')     # Установка не разрешена
    remain = Remain()
```

Данные в управляемых полях name, age и acct вполне ожидаемо поддерживаются для каждого экземпляра (bob остается bob), а остальные тесты проходят, как и ранее:

```
c:\code> py -3 validate_tester2.py validate_descriptors2
[Using: <class 'validate_descriptors2.CardHolder'>]
bob: bob_smith 12345*** 40 123 main st
sue: sue_jones 56781*** 35 124 main st
bob: bob_smith 12345*** 40 123 main st
c:\code> py -3 validate_tester.py validate_descriptors2
...тот же самый вывод, что и в версии со свойствами, кроме имени класса...
```

Одно небольшое предостережение: в имеющемся виде данная версия не поддерживает доступ к дескриптору *через класс*, поскольку в таком случае аргументу экземпляра передается None (также обратите внимание на то, что из-за корректировки имя атрибута __*X* стало выглядеть как _Name__name в сообщении об ошибке, возникающей при попытке извлечения):

```
>>> from validate_descriptors1 import CardHolder
>>> bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
>>> bob.name
'bob_smith'
>>> CardHolder.name
'bob_smith'

>>> from validate_descriptors2 import CardHolder
>>> bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
>>> bob.name
'bob_smith'
>>> CardHolder.name
AttributeError: 'NoneType' object has no attribute '_Name__name'
Ошибка атрибута: объект NoneType не имеет атрибута _Name__name
```

Мы могли бы выявить ситуацию с помощью небольшого объема дополнительного кода, чтобы более явно генерировать ошибку, но вероятно поступать так не имеет смысла. Из-за того, что текущая версия хранит данные в *экземпляре клиентского класса*, дескрипторы ничего не значат, если они не сопровождаются клиентским экземпляром (во многом подобно нормальному несвязанному методу экземпляра). На самом деле в том и заключается весь смысл изменения в этой версии!

Будучи классами, дескрипторы являются удобным и мощным инструментом, но они преподносят варианты, которые способны оказать глубокое влияние на поведение программы. Как всегда в ООП, тщательно выбирайте политики предохранения состояния.

Использование __getattr__ для проверки достоверности

Как мы уже видели, метод __getattr__ перехватывает все неопределенные атрибуты, так что он обеспечивает более обобщенное решение, чем применение свойств и дескрипторов. В нашем примере мы просто проверяем имя атрибута, чтобы выяснить, когда извлекается управляемый атрибут; остальные атрибуты физически хранятся в экземпляре и потому никогда не достигнут __getattr__. Несмотря на бо́льшую универсальность подхода по сравнению со свойствами или дескрипторами, имитирование ориентации на конкретные атрибуты других инструментов может потребовать дополнительной работы. Нам необходимо проверять имена во время выполнения, и мы должны реализовать метод __setattr__ для перехвата и проверки достоверности операций присваивания значений атрибутам.

Что касается версий со свойствами и дескрипторами рассматриваемого примера, то важно обратить внимание, что операции присваиваний значений атрибутам внутри метода конструктора `__init__` тоже запускают метод `__setattr__` класса. Скажем, когда в `__init__` присваивается `self.name`, то автоматически вызывается метод `__setattr__`, который видоизменяет значение и присваивает его атрибуту экземпляра по имени `name`. Хранение `name` в экземпляре гарантирует, что будущие операции доступа не приведут к запуску `__getattr__`. Напротив, `acct` хранится как `_acct`, так что последующие операции доступа к `acct` инициируют вызовы `__getattr__`.

В конечном итоге очередная версия класса, как и предшествующие две, управляет атрибутами `name`, `age` и `acct`, разрешает прямой доступ к атрибуту `addr` и предоставляет атрибут только для чтения по имени `remain`, который является полностью виртуальным и вычисляется по запросу.

Для сравнения эта версия содержит 32 строки кода – на 7 меньше, чем версия на основе свойств и на 13 меньше, чем версия, в которой используются дескрипторы. Конечно, ясность кода важнее его размера, но добавочный код иногда подразумевает дополнительную работу по реализации и сопровождению. Вероятно, здесь более важны *роли*: обобщенные инструменты, подобные `__getattr__`, могут лучше подходить для обобщенного делегирования, в то время как свойства и дескрипторы в большей степени предназначены для управления конкретными атрибутами.

Также обратите внимание, что в коде возникают *дополнительные вызовы* при установке неуправляемых атрибутов (например, `addr`), но такие вызовы отсутствуют при извлечении неуправляемых атрибутов, т.к. они определены. Хотя для большинства программ результатом будут, скорее всего, пренебрежимо малые накладные расходы, более узко сфокусированные *свойства* и *дескрипторы* приводят к дополнительному вызову только при доступе к управляемым атрибутам и также появляются в результатах `dir`, когда в них нуждаются обобщенные инструменты.

Вот версия метода `__getattr__` для проверки достоверности:

```
# Файл validate_getattr.py

class CardHolder:
    acctlen = 8                                 # Данные класса
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                        # Данные экземпляра
        self.name = name                        # Тоже запускают __setattr__
        self.age = age                          # Имя _acct не корректируется:
                                                #   проверяется name
        self.addr = addr                        # Имя addr не является управляемым
                                                # remain не имеет данных
    def __getattr__(self, name):
        if name == 'acct':              # При извлечении неопределенных атрибутов
            return self._acct[:-3] + '***'      # name, age, addr определены
        elif name == 'remain':
            return self.retireage - self.age    # Не запускает __getattr__
        else:
            raise AttributeError(name)

    def __setattr__(self, name, value):
        if name == 'name':             # При операциях присваивания всех атрибутов
           value = value.lower().replace(' ', '_')  # addr хранится напрямую
        elif name == 'age':            # acct корректируется в _acct
```

```
        if value < 0 or value > 150:
            raise ValueError('invalid age')      # недопустимый возраст
    elif name == 'acct':
        name = '_acct'
        value = value.replace('-', '')
        if len(value) != self.acctlen:
          raise TypeError('invald acct number') # недопустимый номер счета
    elif name == 'remain':
        raise TypeError('cannot set remain')
    self.__dict__[name] = value                 # Избегание зацикливания
                                                #   (или через object)
```

При запуске кода с помощью любого из двух тестовых сценариев получается тот же самый вывод (с другим именем класса):

```
c:\code> py -3 validate_tester.py validate_getattr
...тот же самый вывод, что и в версии со свойствами, кроме имени класса...
c:\code> py -3 validate_tester2.py validate_getattr
...тот же самый вывод, что и в версии с дескрипторами состояния экземпляра,
кроме имени класса...
```

Использование __getattribute__ для проверки достоверности

В финальном варианте применяется метод `__getattribute__` для перехвата операций извлечения атрибутов и управления ими надлежащим образом. Здесь перехватывается извлечение каждого атрибута, так что мы проверяем имена атрибутов, чтобы обнаруживать управляемые атрибуты и направлять все остальные суперклассу для нормальной обработки операций извлечения. Для перехвата операций присваивания данная версия использует тот же самый метод `__setattr__`, что и предыдущая версия.

Код работает очень похоже на версию `__getattr__`, поэтому полное описание здесь не повторяется. Однако обратите внимание, что поскольку методу `__getattribute__` направляется операция извлечения *каждого* атрибута, нам нет нужды корректировать имена для их перехвата (`acct` хранится как `acct`). С другой стороны, код обязан позаботиться о направлении операций извлечения неуправляемых атрибутов суперклассу во избежание зацикливания или дополнительных вызовов.

Также имейте в виду, что в этой версии возникают дополнительные вызовы для установки и извлечения неуправляемых атрибутов (скажем, `addr`); если первостепенным требованием является скорость, тогда текущая альтернатива может оказаться самой медленной в наборе. Для сравнения данная версия содержит 32 строки кода, как и предыдущая версия, и включает необходимое наследование от `object` для совместимости с Python 2.X; подобно свойствам и дескрипторам метод `__getattribute__` представляет собой инструмент классов нового стиля:

```
# Файл validate_getattribute.py

class CardHolder(object):            # В Python 2.X требуется (object)
    acctlen = 8                      # Данные класса
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct             # Данные экземпляра
```

```
        self.name = name              # Тоже запускают __setattr__
        self.age = age                # Имя _acct не корректируется:
                                      #   проверяется name
        self.addr = addr              # Имя addr не является управляемым
                                      # remain не имеет данных
    def __getattribute__(self, name):
        superget = object.__getattribute__    # Зацикливания нет:
                                              #   на один уровень выше
        if name == 'acct':                    # При извлечении всех атрибутов
            return superget(self, 'acct')[:-3] + '***'
        elif name == 'remain':
            return superget(self, 'retireage') - superget(self, 'age')
        else:
            return superget(self, name)       # name, age, addr: хранятся

    def __setattr__(self, name, value):
        if name == 'name':          # При операциях присваивания всех атрибутов
           value = value.lower().replace(' ', '_') # addr хранится напрямую
        elif name == 'age':
            if value < 0 or value > 150:
                raise ValueError('invalid age')     # недопустимый возраст
        elif name == 'acct':
            value = value.replace('-', '')
            if len(value) != self.acctlen:
              raise TypeError('invald acct number') # недопустимый номер счета
        elif name == 'remain':
            raise TypeError('cannot set remain')
        self.__dict__[name] = value  # Избегание зацикливания, исходные имена
```

При запуске с обоими тестовыми сценариями в Python 2.X или 3.X версии `getattr*` и `getattribute*` работают точно так же, как версии со свойствами и дескрипторами для каждого экземпляра клиентского класса. Они демонстрируют *четыре способа достижения той же самой цели в Python*, хотя имеют отличающиеся структуры и вероятно менее избыточны в ряде других ролей. Обязательно изучите и запустите код из настоящего раздела самостоятельно, чтобы получить лучшее представление о методиках реализации управляемых атрибутов.

Резюме

В главе были раскрыты разнообразные приемы управления доступом к атрибутам в Python, включая методы перегрузки операций `__getattr__` и `__getattribute__`, свойства классов и дескрипторы атрибутов классов. Попутно было проведено сравнение рассматриваемых инструментов и предложено несколько сценариев использования для демонстрации их поведения.

В главе 39 обзор средств для построения инструментов продолжается исследованием *декораторов* — кода, который автоматически запускается во время создания функций и классов, а не при доступе к атрибутам. Но прежде чем переходить к ее чтению, ответьте на контрольные вопросы главы, чтобы закрепить полученные знания.

Проверьте свои знания: контрольные вопросы

1. Чем между собой отличаются `__getattr__` и `__getattribute__`?
2. Чем между собой отличаются свойства и дескрипторы?
3. Как связаны друг с другом свойства и декораторы?
4. Каковы главные функциональные отличия между `__getattr__` и `__getattribute__`, а также между свойствами и дескрипторами?
5. Разве все это сравнение возможностей — не просто разновидность спора?

Проверьте свои знания: ответы

1. Метод `__getattr__` запускается только для операций извлечения *неопределенных* атрибутов (т.е. тех, которые не присутствуют в экземпляре и не наследуются из любого его класса). По контрасту с ним метод `__getattribute__` вызывается для операции извлечения *каждого* атрибута, определен он или нет. По этой причине код внутри `__getattr__` может свободно извлекать другие атрибуты, если они определены, тогда как в методе `__getattribute__` для извлечения таких атрибутов должен использоваться специальный код, чтобы избежать зацикливания или дополнительных вызовов (он обязан направлять операции извлечения суперклассу, пропуская себя).
2. Свойства исполняют особую роль, в то время как дескрипторы более универсальны. Свойства определяют функции извлечения, установки и удаления для конкретного атрибута; дескрипторы снабжают класс методами для таких действий, но предоставляют добавочную гибкость с целью поддержки более произвольных действий. На самом деле свойства являются простым способом создания специфического вида дескриптора — такого, который запускает функции при доступе к атрибутам. Реализация тоже отличается: свойство создается с помощью встроенной функции, а дескриптор — посредством класса; таким образом, дескрипторы могут воспользоваться в своих интересах всеми обычными возможностями ООП, касающимися классов, вроде наследования. Кроме того, вдобавок к информации состояния экземпляра дескрипторы имеют собственное локальное состояние, так что временами они способны избегать конфликтов имен в экземпляре.
3. Свойства могут быть реализованы с помощью декораторного синтаксиса. Поскольку встроенная функция `property` принимает единственный аргумент типа функции, она может использоваться напрямую как декоратор функции для определения свойства с доступом по извлечению. Благодаря поведению декораторов, предусматривающему повторную привязку имен, имя декорированной функции присваивается свойству, чей метод извлечения устанавливается в исходную функцию (`name = property(name)`). Атрибуты `setter` и `deleter` свойства позволяют дополнительно добавлять методы установки и удаления посредством декораторного синтаксиса — они устанавливают метод доступа в декорированную функцию и возвращают дополненное свойство.

4. Методы __getattr__ и __getattribute__ являются более обобщенными: они могут применяться для перехвата произвольно большого количества атрибутов. В противоположность им каждое свойство или дескриптор обеспечивают перехват доступа только для одного *конкретного* свойства – перехватывать операцию извлечения каждого атрибута с помощью единственного свойства или дескриптора не удастся. С другой стороны, свойства и дескрипторы изначально обрабатывают операции извлечения и *присваивания* для атрибута: __getattr__ и __getattribute__ обрабатывают только операции извлечения; чтобы перехватывать также операции присваивания, потребуется реализовать еще и метод __setattr__. Реализация тоже отличается: __getattr__ и __getattribute__ являются методами перегрузки операций, тогда как свойства и дескрипторы представляют собой объекты, вручную присвоенные атрибутам класса. В отличие от остальных свойства и дескрипторы способны иногда избегать дополнительных вызовов при присваивании неуправляемых имен плюс автоматически отображаются в результатах dir, но пределы их возможностей более узкие – они не могут достичь целей обобщенной координации вызовов. С развитием Python новые средства обычно предлагают альтернативы, но не полностью соответствуют тому, что было раньше.

5. Нет, это не так. Ниже дается вольная интерпретация дискуссии из скетча "Летающий цирк Монти Пайтона".

```
Спор - это ряд взаимосвязанных доводов, призванных отстоять свою позицию.
Нет, это не так.
Да, это так! Это не просто отрицание.
Послушай, если я спорю с тобой, то должен занять противоположную позицию.
Да, но это не значит просто сказать: " Нет, это не так".
Да, это так!
Нет, это не так!
Да, это так!
Нет, это не так. Спор - это интеллектуальный процесс. Отрицание -
всего лишь возражение на любой довод, приводимый другим человеком.
(после короткой паузы) Нет, это не так.
Это так.
Вовсе нет.
А теперь послушай...
```

ГЛАВА 39

Декораторы

В главе 32, посвященной расширенным возможностям классов, мы встречались со статическими методами и методами классов, кратко рассмотрели декораторный синтаксис @, предлагаемый Python для их объявления, и предварительно ознакомились с методиками реализации декораторов. Декораторы функций также бегло упоминались в главе 38 при исследовании способности встроенной функции `property` выступать в качестве декоратора и в главе 29 во время изучения понятия абстрактных суперклассов.

В этой главе раскрытие декораторов продолжается с того места, где оно было оставлено ранее. Здесь мы подробнее обсудим внутреннюю работу декораторов и ознакомимся с более развитыми способами реализации новых декораторов. Как выяснится, в декораторах будут регулярно обнаруживаться многие концепции, которые исследовались ранее, особенно предохранение состояния.

Тема довольно-таки сложная, а создание декораторов больше интересует скорее разработчиков инструментов, чем прикладных программистов. Тем не менее, учитывая все возрастающее распространение декораторов в популярных фреймворках для Python, их базовое понимание может помочь в прояснении исполняемой ими роли, даже если вы являетесь обычным пользователем декораторов.

Помимо описания деталей создания декораторов настоящая глава служит более реалистичным учебным пособием по применению Python. Поскольку приводимые в ней примеры оказываются крупнее, чем в большинстве других глав книги, они лучше иллюстрируют способы объединения кода в более завершенные системы и инструменты. В качестве дополнительного преимущества некоторые написанные здесь программы могут использоваться как универсальные инструменты при повседневном программировании.

Что такое декоратор?

Декорирование представляет собой способ указания управляющего или дополняющего кода для функций и классов. Сами декораторы принимают вид вызываемых объектов (например, функций), обрабатывающих другие вызываемые объекты. Как было показано ранее в книге, декораторы Python имеют две связанные друг с другом формы, ни одна из которых не требует Python 3.X или классов нового стиля.

- Декораторы функций, добавленные в Python 2.4, делают повторное привязывание имен во время определения функций, предоставляя уровень логики, который может управлять функциями и методами или последующими обращениями к ним.

- Декораторы классов, добавленные в Python 2.6 и 3.0, делают повторное привязывание имен во время определения классов, предоставляя уровень логики, который может управлять классами или экземплярами, созданными при последующих обращениях к классам.

Выражаясь кратко, декораторы предлагают способ вставки автоматически запускаемого кода в конце операторов определения функций и классов — в конце `def` для декораторов функций и в конце `class` для декораторов классов. Такой код может исполнять множество ролей, как будет описано в дальнейших разделах.

Управление вызовами и экземплярами

В типичной ситуации такой автоматически запускаемый код может применяться для дополнения обращений к функциям и классам. Это организуется за счет ввода в действие объектов оболочек (известных как *посредники*), предназначенных для вызова в более позднее время.

Посредники вызовов

Декораторы функций вводят в действие объекты оболочек, которые позволяют перехватывать последующие *вызовы функций* и обрабатывать их по мере надобности, обычно передавая сами вызовы исходной функции для выполнения управляемого действия.

Посредники интерфейсов

Декораторы классов вводят в действие объекты оболочек, которые позволяют перехватывать последующие *вызовы для создания экземпляров* и при необходимости обрабатывать их, обычно передавая сами вызовы исходному классу для создания управляемого экземпляра.

Декораторы достигают таких эффектов за счет автоматической повторной привязки имен функций и классов к другим вызываемым объектам в конце операторов `def` и `class`. При обращении в более позднее время привязанные вызываемые объекты могут выполнять задачи наподобие отслеживания и измерения времени вызовов функций, управления доступом к атрибутам экземпляров и т.д.

Управление функциями и классами

Хотя большинство примеров в главе имеют дело с использованием объектов оболочек для перехвата последующих обращений к функциям и классам, это не единственный способ применения декораторов.

Администраторы функций

Декораторы функций также могут использоваться для управления *объектами функций* взамен или в дополнение к их последующим вызовам, скажем, чтобы регистрировать функции в API-интерфейсах. Однако основное внимание здесь будет уделяться их более распространенному применению в качестве оболочек вызовов.

Администраторы классов

Декораторы классов также могут использоваться для непосредственного управления *объектами классов* взамен или в дополнение к вызовам, создающим экземпляры, например, чтобы дополнить класс новыми методами. Поскольку такая роль плотно пересекается с ролью *метаклассов*, в следующей главе будут приве-

дены дополнительные сценарии применения. Как выяснится, оба инструмента запускаются в конце процесса создания классов, но декораторы классов часто предлагают более легковесное решение.

Другими словами, декораторы функций могут использоваться для управления вызовами функций и объектами функций, а декораторы классов — для управления экземплярами классов и собственно классами. За счет возвращения самого декорируемого объекта вместо оболочки декораторы становятся для функций и классов простым шагом, предпринимаемым после создания.

Независимо от роли, которую исполняют декораторы, они обеспечивают удобный и явный способ реализации инструментов, полезный на стадии разработки программ и в действующих производственных системах.

Использование и определение декораторов

В зависимости от вашего рабочего задания вы можете иметь дело с декораторами как пользователь или как поставщик (вы также могли бы заниматься сопровождением, но это лишь означает принятие нейтральной позиции). Мы увидим, что в состав Python входят встроенные декораторы, исполняющие специализированные роли — объявление статических методов и методов классов, создание свойств и т.д. Вдобавок многие популярные инструментальные комплекты для Python содержат декораторы, предназначенные для решения таких задач, как управление логикой работы с базами данных или пользовательскими интерфейсами. В подобных случаях мы вполне можем обойтись без знания о том, как реализуются декораторы.

Для более общих задач программисты могут самостоятельно создавать произвольные декораторы. Скажем, декораторы функций могут применяться для дополнения функций кодом, который отслеживает или регистрирует вызовы в журнале, производит проверку допустимости аргументов на стадии отладки, автоматически получает и освобождает блокировки в потоках, измеряет время выполнения вызовов функций в целях оптимизации и делает многое другое. Любое мыслимое поведение, которое вы пожелали бы добавить к вызову функции (в действительности поместив его внутрь оболочки), является кандидатом для специальных декораторов функций.

С другой стороны, декораторы функций предназначены для дополнения только *вызовов* конкретных функций или методов, а не всего *объектного интерфейса*. С последней ролью лучше справляются декораторы классов — учитывая возможность перехвата ими вызовов, создающих экземпляры, их можно использовать для реализации любых дополнений объектных интерфейсов или задач управления. Например, специальные декораторы классов способны отслеживать, проверять допустимость либо иным образом дополнять каждую ссылку на атрибут, предпринятую для объекта. Они также могут применяться для реализации объектов-посредников, классов-одиночек и других общих кодовых схем. На самом деле мы обнаружим, что многие декораторы классов сильно напоминают — и фактически являются главным приложением — кодовой схемы делегирования, которая обсуждалась в главе 31.

Для чего используются декораторы?

Подобно многим расширенным инструментам Python с чисто технической точки зрения декораторы никогда не считаются строго обязательными: мы часто в состоянии реализовать их функциональность с использованием вызовов простых вспомогательных функций или других приемов. И на базовом уровне мы всегда можем вручную написать код повторной привязки имен, которую декораторы выполняют автоматически.

Тем не менее, декораторы предлагают для таких задач явный синтаксис, который проясняет намерение, способен минимизировать избыточность дополняющего кода и может содействовать в обеспечении корректного применения API-интерфейсов.

- Декораторы имеют чрезвычайно явный синтаксис, что позволяет заметить их гораздо быстрее, чем вызовы вспомогательных функций, которые могут находиться произвольно далеко от целевых функций или классов.
- Декораторы применяются один раз при определении целевой функции или класса; нет необходимости снабжать каждое обращение к классу или функции дополнительным кодом, который может потребовать внесения изменений в будущем.
- Учитывая оба предшествующих пункта, декораторы снижают вероятность того, что пользователь API-интерфейса забудет дополнить функцию или класс в соответствии с требованиями данного API-интерфейса.

Другими словами, помимо своей специальной модели декораторы предлагают ряд преимуществ в плане сопровождения и согласованности кода. Кроме того, как инструменты структурирования, декораторы естественным образом содействуют *инкапсуляции* кода, которая сокращает избыточность и облегчает внесение изменений в будущем.

Декораторы также обладают потенциальными *недостатками* – когда они вставляют логику оболочки, то могут изменить типы декорированных объектов и повлечь за собой дополнительные вызовы в случае использования в качестве посредников для вызовов либо интерфейсов. С другой стороны, те же соображения применимы к любой методике добавления логики оболочки к объектам.

Позже в главе мы исследуем такие компромиссы в контексте реального кода. Несмотря на то что решение использовать декораторы все же в чем-то субъективно, их преимущества достаточно убедительны для того, чтобы они быстро стали рекомендованной практикой в мире Python. Давайте рассмотрим детали, которые помогут вам сделать собственный выбор.

Декораторы или макросы. Декораторы Python имеют сходство с тем, что в других языках называется *аспектно-ориентированным программированием*, которое предусматривает вставку кода, подлежащего автоматическому запуску до и после выполнения вызова функции. Синтаксис декораторов также очень близко напоминает синтаксис *аннотаций* Java (и вероятно позаимствован оттуда), хотя модель Python, как правило, считается более гибкой и универсальной.

Некоторые сравнивают декораторы и с *макросами*, но это не совсем уместно и может даже вводить в заблуждение. Макросы (скажем, директива препроцессора `#define` в языке C) обычно ассоциируются с текстовой заменой и расширением и предназначены для генерации кода. Наоборот, декораторы Python являются операциями *времени выполнения*, основанными на повторной привязке имен, вызываемых объектах и зачастую посредниках. Наряду с тем, что декораторы и макросы могут иметь временами перекрывающиеся сценарии применения, они фундаментально отличаются областью действия, реализацией и кодовыми схемами. Их сравнение сродни сравнению оператора `import` в Python с директивой `#include` в C, когда похожим образом путаются основанная на объектах операция времени выполнения и вставка текста.

Конечно, со временем термин *макрос* был несколько ослаблен – для некоторых теперь он может также означать любую заготовленную последовательность шагов или процедуру – и пользователи других языков могут счесть аналогию с дескрипторами в любом случае полезной. Но им, вероятно, следует иметь в виду и то, что декораторы имеют дело с вызываемыми *объектами*, которые управляют вызываемыми *объектами*, а не с расширением текста. Язык Python, как правило, лучше постигать и использовать в переводе на идиомы Python.

Основы

Давайте начнем с того, что рассмотрим поведение декорирования в фигуральном смысле. Вскоре мы напишем реальный и более содержательный код, но поскольку большая часть магии декораторов сводится к автоматической операции повторной привязки, важно сначала понять такое соответствие.

Декораторы функций

Декораторы функций были доступны, начиная с версии Python 2.4. Как упоминалось ранее в книге, они в основном представляют собой всего лишь синтаксический сахар, который обеспечивает запуск одной функции через другую в конце оператора def и повторно привязывает имя исходной функции к результату.

Использование

Декоратор функции является своего рода *объявлением времени выполнения* о функции, чье определение следует за декоратором. Декоратор записывается в строке прямо перед оператором def, определяющим функцию или метод, и состоит из символа @, за которым находится ссылка на *метафункцию* – функцию (или другой вызываемый объект), управляющую другой функцией.

В переводе на код декораторы функций автоматически отображают показанный ниже синтаксис:

```
@decorator              # Декорирование функции
def F(arg):
    ...

F(99)                   # Вызов функции
```

на следующую эквивалентную форму, где decorator – это вызываемый объект с одним аргументом, который возвращает вызываемый объект с таким же количеством аргументов, как у функции F (если не саму F):

```
def F(arg):
    ...
F = decorator(F)        # Повторная привязка имени функции к результату декоратора
F(99)                   # По существу вызывается decorator(F)(99)
```

Такая автоматическая повторная привязка имен работает с любым оператором def, определяет он простую функцию или метод внутри класса. Когда функция F позже вызывается, фактически производится обращение к объекту, *возвращенному* декоратором, который может быть либо другим объектом, реализующим необходимую логику оболочки, либо самой исходной функцией.

Другими словами, декорирование по существу отображает показанное далее первое выражение на второе — хотя декоратор в действительности выполняется только раз во время декорирования:

```
func(6, 7)
decorator(func)(6, 7)
```

Автоматическая повторная привязка имен объясняет синтаксис декорирования статических методов и свойств, встречавшийся ранее в книге:

```
class C:
   @staticmethod
   def meth(...): ...         # meth = staticmethod(meth)

class C:
   @property
   def name(self): ...        # name = property(name)
```

В обоих случаях имя метода повторно привязывается к результату встроенного декоратора в конце оператора def. Дальнейший вызов исходного имени инициирует обращение к любому объекту, который возвратил декоратор. В приведенных особых случаях исходные имена повторно привязываются к статическому методу и дескриптору свойства, но как объясняется в следующем разделе, процесс намного более универсален.

Реализация

Сам декоратор является *вызываемым объектом, возвращающим вызываемый объект*. То есть он возвращает объект, к которому производится обращение позже, когда декорированная функция вызывается через свое исходное имя — либо объект оболочки для перехвата будущих вызовов, либо исходную функцию, дополненную каким-нибудь образом. На самом деле декораторы способны *быть* вызываемым объектом любого типа и *возвращать* вызываемый объект любого типа: может применяться любое сочетание функций и классов, хотя некоторые лучше подходят в определенных контекстах.

Например, чтобы использовать протокол декорирования для управления функцией сразу же после ее создания, мы могли бы реализовать декоратор такого вида:

```
def decorator(F):
   # Обработка функции F
   return F

@decorator
def func(): ...              # func = decorator(func)
```

Поскольку исходная декорированная функция снова присваивается своему имени, в итоге к определению функции просто добавляется шаг после создания. Структура подобного рода может применяться для регистрации функции в API-интерфейса, присваивания атрибутов функций и т.д.

В более типичной ситуации для вставки логики перехвата последующих обращений к функции мы могли бы реализовать декоратор, который возвращает объект, отличающийся от исходной функции — посредник для вызовов в более позднее время:

```
def decorator(F):
   # Сохранение либо использование функции F
   # Возвращение другого вызываемого объекта:
   #   вложенного def, класса с __call__ и т.д.
@decorator
def func(): ...              # func = decorator(func)
```

Декоратор вызывается во время декорирования, а вызываемый объект, который он возвращает, вызывается при обращении к исходному имени функции в будущем. Сам декоратор принимает декорированную функцию; возвращенный вызываемый объект принимает любые аргументы, переданные позже имени декорированной функции. При надлежащей реализации все работает аналогично *методам* уровня класса: в первом аргументе возвращенного вызываемого объекта оказывается подразумеваемый объект экземпляра.

Ниже показана общая кодовая схема, воплощающая эту идею — декоратор возвращает объект-оболочку, который предохраняет исходную функцию в объемлющей области видимости:

```
def decorator(F):                # При декорировании @
    def wrapper(*args):          # При вызове внутренней функции
        # Использование функции F и аргументов
        # F(*args) вызывает исходную функцию
    return wrapper

@decorator                       # func = decorator(func)
def func(x, y):                  # func передается функции F декоратора
    ...

func(6, 7)                       # 6, 7 передается аргументу *args оболочки
```

Когда позже происходит обращение к имени `func`, в действительности вызывается функция `wrapper`, возвращенная декоратором `decorator`; функция `wrapper` затем может запустить исходную функцию `func`, т.к. она по-прежнему доступна в *объемлющей области видимости*. При реализации подобным образом каждая декорированная функция производит новую область видимости для предохранения состояния.

Чтобы сделать то же самое посредством *классов*, мы можем перегрузить операцию вызова и применять атрибуты экземпляра вместо объемлющих областей видимости:

```
class decorator:
    def __init__(self, func):  # При декорировании @
        self.func = func
    def __call__(self, *args): # При вызове внутренней функции
        # Использование self.func и аргументов
        # self.func(*args) вызывает исходную функцию

@decorator
def func(x, y):                  # func = decorator(func)
    ...                          # func передается __init__

func(6, 7)                       # 6, 7 передается аргументу *args метода __call__
```

Когда позже происходит обращение к имени `func`, то в действительности вызывается метод перегрузки операций `__call__` экземпляра, созданного декоратором `decorator`; метод `__call__` затем запускает исходную функцию `func`, потому что она по-прежнему доступна в *атрибуте экземпляра*. В случае такой реализации каждая декорированная функция выпускает новый экземпляр для предохранения состояния.

Поддержка декорирования методов

С предыдущей реализацией, основанной на классе, связан один тонкий момент. Несмотря на то что она нормально работает при перехвате вызовов простых *функций*, этого не происходит в случае ее применения к функциям *методов* на уровне класса:

```
class decorator:
   def __init__(self, func):      # func - метод без экземпляра
      self.func = func
   def __call__(self, *args):     # self - экземпляр декоратора

      # self.func(*args) терпит неудачу!
      # Экземпляр C не находится в args!

class C:
   @decorator
   def method(self, x, y):        # method = decorator(method)
      ...                         # Повторная привязка к экземпляру декоратора
```

При такой реализации декорированный метод повторно привязывается к экземпляру класса декоратора вместо простой функции.

Проблема заключается в том, что аргумент `self` в методе `__call__` декоратора получает экземпляр класса `decorator`, когда метод вызывается позже, и экземпляр класса `C` никогда не помещается в `*args`. В результате направление вызова исходному методу становится невозможным — объект декоратора хранит исходную функцию метода, но не располагает экземпляром, чтобы ей передать.

Для поддержки *и* функций, *и* методов лучше подойдет альтернатива в виде вложенной функции:

```
def decorator(F):            # F - функция или метод без экземпляра
   def wrapper(*args):       # Для метода экземпляр класса находится в args[0]

      # F(*args) запускает func или method

   return wrapper

@decorator
def func(x, y):              # func = decorator(func)
   ...
   func(6, 7)                # В действительности вызывается wrapper(6, 7)

class C:
   @decorator
   def method(self, x, y):   # method = decorator(method)
      ...                    # Повторная привязка к простой функции

X = C()
X.method(6, 7)               # В действительности вызывается wrapper(X, 6, 7)
```

Здесь метод `wrapper` принимает в своем первом аргументе экземпляр класса `C`, поэтому он может направляться на исходный метод и получать доступ к информации состояния.

Формально такая версия с вложенной функцией работает из-за того, что Python создает объект связанного метода и потому передает экземпляр целевого класса аргументу `self`, только когда атрибут метода ссылается на простую функцию. Когда взамен он ссылается на экземпляр вызываемого класса, то данный экземпляр передается в `self`, чтобы предоставить вызываемому классу доступ к собственной информации состояния. Мы увидим, что это тонкое отличие может иметь значение в более реалистичных примерах, рассматриваемых позже в главе.

Также обратите внимание, что вложенные функции являются, пожалуй, самым прямолинейным способом поддержки декорирования функций и методов, но отнюдь не единственным. Скажем, *дескрипторы* из предыдущей главы при вызове получают экземпляры дескриптора и целевого класса.

Несмотря на более высокую сложность, далее в главе будет показано, каким образом задействовать данный инструмент и в таком контексте.

Декораторы классов

Декораторы функций настолько доказали свою полезность, что модель была расширена для поддержки декорирования классов, начиная с версий Python 2.6 и 3.0. Поначалу декораторам классов оказывалось сопротивление, т.к. их роль частично совпадала с ролью *метаклассов*; однако, в конце концов, их официально приняли из-за обеспечения ими более простого способа для достижения многих тех же целей.

Декораторы классов тесно связаны с декораторами функций; по сути, они используют одинаковый синтаксис и очень похожие кодовые схемы. Тем не менее, вместо помещения в оболочку индивидуальных функций или методов декораторы классов управляют классами или снабжают вызовы, создающие экземпляры, дополнительной логикой, которая управляет экземплярами, созданными из класса, или дополняет их. Во второй роли они могут управлять полными объектными интерфейсами.

Использование

Синтаксически декораторы классов указываются прямо перед операторами `class` в той же манере, как декораторы функций находятся непосредственно перед операторами `def`. Формально для декоратора `decorator`, который обязан быть вызываемым объектом, принимающим один аргумент и возвращающий вызываемый объект, показанный ниже синтаксис декоратора классов:

```
@decorator              # Декорирование класса
class C:
    ...
x = C(99)               # Создание экземпляра
```

эквивалентен следующему — класс автоматически передается функции декоратора, а ее результат присваивается имени класса:

```
class C:
    ...
C = decorator(C)        # Повторная привязка имени класса к результату декоратора
x = C(99)               # По существу вызывается decorator(C)(99)
```

Совокупный эффект заключается в том, что обращение в будущем к имени класса для создания экземпляра приводит к запуску возвращенного декоратором вызываемого объекта, который может обращаться к самому исходному классу или нет.

Реализация

Новые декораторы классов реализуются с помощью многих тех же методик, которые применялись с декораторами функций, хотя часть их могут включать в себя *два уровня* дополнения — для управления как вызовами, создающими экземпляры, так и доступом к интерфейсу экземпляра. Поскольку декоратор классов также является *вызываемым объектом, который возвращает вызываемый объект*, его будет вполне достаточно для большинства сочетаний функций и классов.

Однако как бы декоратор классов ни был реализован, его результат запускается при последующем создании экземпляра.

Например, для простого управления классом сразу после его создания понадобится возвращать сам исходный класс:

```
def decorator(C):
    # Обработка класса C
    return C

@decorator
class C: ...                              # C = decorator(C)
```

Чтобы взамен вставить уровень оболочки, который перехватывает будущие вызовы, создающие экземпляры, необходимо возвращать другой вызываемый объект:

```
def decorator(C):
    # Сохранение либо использование класса C
    # Возвращение другого вызываемого объекта:
    #   вложенного def, класса с __call__ и т.д.

@decorator
class C: ...                              # C = decorator(C)
```

Вызываемый объект, возвращаемый таким декоратором классов, обычно создает и возвращает новый экземпляр исходного класса, каким-то образом дополненный для управления его интерфейсом. Скажем, следующий декоратор вставляет объект, который перехватывает доступ к неопределенным атрибутам экземпляра класса:

```
def decorator(cls):                       # При декорировании @
    class Wrapper:
        def __init__(self, *args):        # При создании экземпляров
            self.wrapped = cls(*args)
        def __getattr__(self, name):      # При извлечении атрибутов
            return getattr(self.wrapped, name)
    return Wrapper

@decorator
class C:                                  # C = decorator(C)
    def __init__(self, x, y):             # Запускается методом Wrapper.__init__
        self.attr = 'spam'

x = C(6, 7)          # В действительности вызывается Wrapper(6, 7)
print(x.attr)        # Запускается Wrapper.__getattr__, выводится spam
```

В приведенном примере декоратор повторно привязывает имя класса к другому классу, который предохраняет исходный класс в объемлющей области видимости и при обращении к исходному классу создает и внедряет его экземпляр. Когда позже из экземпляра извлекается какой-нибудь атрибут, операция перехватывается методом `__getattr__` объекта-оболочки и ее выполнение делегируется внедренному экземпляру исходного класса. Кроме того, каждый декорированный класс создает новую область видимости, которая запоминает исходный класс. Позже в главе мы расширим этот пример, превратив его в более полезный код.

Подобно декораторам функций декораторы классов обычно реализуются в виде "фабричных" функций, создающих и возвращающих вызываемые объекты, в форме классов, которые используют методы `__init__` или `__call__` для перехвата операций вызова, либо в виде какой-то их комбинации. Фабричные функции, как правило, предохраняют состояние в ссылках из объемлющих областей видимости, а классы — в атрибутах.

Поддержка множества экземпляров

Как и с декораторами функций, для декораторов классов одни комбинации вызываемых типов работают лучше, чем другие. Рассмотрим следующую ошибочную альтернативу для декоратора классов из предыдущего примера:

```
class Decorator:
    def __init__(self, C):                # При декорировании @
        self.C = C
    def __call__(self, *args):            # При создании экземпляров
        self.wrapped = self.C(*args)
        return self
    def __getattr__(self, attrname):      # При извлечении атрибутов
        return getattr(self.wrapped, attrname)
@Decorator
class C: ...                              # C = Decorator(C)

x = C()
y = C()                                   # Переписывает x!
```

В коде обрабатывается множество декорированных классов (каждый создает новый экземпляр `Decorator`) и перехватываются вызовы, создающие экземпляры (каждый запускает метод `__call__`). Тем не менее, в отличие от предыдущей показанная выше версия терпит неудачу при обработке *множества экземпляров* заданного класса — каждый вызов, создающий экземпляр, переписывает ранее сохраненный экземпляр. Исходная версия не поддерживает множество экземпляров, потому что каждый вызов, создающий экземпляр, создает новый независимый объект-оболочку. В более общем случае любая из следующих схем поддерживает множество внутренних экземпляров:

```
def decorator(C):                     # При декорировании @
    class Wrapper:
      def __init__(self, *args):  # При создании экземпляров: новый объект Wrapper
            self.wrapped = C(*args)   # Внедрение экземпляра в экземпляр
    return Wrapper

class Wrapper: ...
def decorator(C):                    # При декорировании @
    def onCall(*args):               # При создании экземпляров: новый объект Wrapper
      return Wrapper(C(*args))       # Внедрение экземпляра в экземпляр
    return onCall
```

Позже в главе мы исследуем это явление в более реалистичном контексте; однако, на практике мы обязаны позаботиться о надлежащем комбинировании вызываемых типов для поддержки нашего намерения и разумном выборе политик предохранения состояния.

Вложение декораторов

Временами одного декоратора недостаточно. Например, предположим, что вы реализовали *два* декоратора функций, предназначенных для применения на стадии разработки — один для проверки типов аргументов перед вызовом функции и еще один для проверки типа возвращаемого значения после вызова функции. Вы можете использовать любой из них независимо, но что делать, если желательно задействовать *оба* декоратора в одиночной функции? В действительности вам необходим способ *вложения* декораторов, чтобы результат одного декоратора был функцией, декорированной другим декоратором. До тех пор, пока при последующих вызовах выполняются оба шага, совершенно не важно, какой из декораторов будет вложенным.

Для поддержки множества вложенных шагов дополнения декораторный синтаксис позволяет добавлять к декорированной функции или методу несколько уровней логики оболочки. Когда применяется такая возможность, каждый декоратор должен находиться в собственной строке. Декораторный синтаксис в форме:

```
@A
@B
@C
def f(...):
    ...
```

выполняется аналогично такому синтаксису:

```
def f(...):
    ...
f = A(B(C(f)))
```

Здесь исходная функция проходит через три разных декоратора, а результирующий вызываемый объект присваивается исходному имени. Каждый декоратор обрабатывает результат предыдущего декоратора, который может быть исходной функцией или вставленным объектом-оболочкой.

Если все декораторы вставляют объекты-оболочки, тогда совокупный эффект заключается в том, что при обращении к имени исходной функции будут вызываться три разных уровня оболочки для дополнения исходной функции тремя разными способами. Декоратор, указанный последним, применяется первым и является наиболее глубоко вложенным декоратором, когда позже происходит вызов имени исходной функции.

Как и в случае с декораторами функций, множество декораторов классов дают в результате множество вызовов вложенных функций и возможно множество уровней и шагов логики оболочки вокруг вызовов, создающих экземпляры. Скажем, следующий код:

```
@spam
@eggs
class C:
    ...
X = C()
```

эквивалентен такому коду:

```
class C:
    ...
C = spam(eggs(C))
X = C()
```

И снова каждый декоратор волен возвращать либо исходный класс, либо вставленный объект-оболочку. Благодаря объектам-оболочкам, когда в конечном итоге запрашивается экземпляр исходного класса C, вызов перенаправляется объектам уровня оболочки, предоставляемым декораторами spam и eggs, которые способны исполнять совершенно разные роли — например, они могут отслеживать и проверять допустимость доступа к атрибутам, причем оба шага выполнялись бы при будущих запросах.

Скажем, приведенные ниже ничего не делающие декораторы просто возвращают декорированную функцию:

```
def d1(F): return F
def d2(F): return F
def d3(F): return F
```

```
@d1
@d2
@d3
def func():                     # func = d1(d2(d3(func)))
    print('spam')

func()                          # Выводится spam
```

С классами работает такой же синтаксис, как у этих ничего не делающих декораторов.

Тем не менее, когда декораторы вставляют объекты функций оболочки, они могут дополнять исходную функцию при вызове — следующий код выполняет конкатенацию с ее результатом на уровнях декораторов по мере провождения уровней от внутреннего к внешнему:

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3
def func():                     # func = d1(d2(d3(func)))
    return 'spam'

print(func())                   # Выводится XYZspam
```

Для реализации уровней оболочки мы используем функции `lambda` (каждая предохраняет внутреннюю функцию в объемлющей области видимости); на практике объекты-оболочки могут принимать форму функций, вызываемых классов и т.д. При надлежащем проектировании вложение декораторов позволяет комбинировать шаги дополнения разнообразными способами.

Аргументы декораторов

Декораторы функций и классов также могут выглядеть как принимающие *аргументы*, хотя фактически эти аргументы передаются вызываемому объекту, который в действительности *возвращает* декоратор, а тот в свою очередь возвращает вызываемый объект. В итоге обычно устанавливается множество уровней предохранения состояния. Например, следующий код:

```
@decorator(A, B)
def F(arg):
    ...

F(99)
```

автоматически отображается на показанную ниже эквивалентную форму, где `decorator` представляет собой вызываемый объект, который *возвращает* действительный декоратор. Возвращенный декоратор в свою очередь возвращает вызываемый объект, запускаемый позже для обращений к имени исходной функции:

```
def F(arg):
    ...
F = decorator(A, B)(F)  # Повторная привязка F к возвращаемому значению decorator

F(99)                   # По существу вызывается decorator(A, B)(F)(99)
```

Аргументы декораторов распознаются до того, как происходит декорирование, и обычно применяются для предохранения информации состояния с целью использо-

вания в будущих вызовах. Скажем, функция декоратора в рассматриваемом примере может принимать такую форму:

```
def decorator(A, B):
    # Сохранение либо использование A, B
    def actualDecorator(F):
      # Сохранение либо использование функции F
      # Возвращение вызываемого объекта: вложенного def, класса с __call__ и т.д.
      return callable
    return actualDecorator
```

Внешняя функция в этой структуре обычно сохраняет аргументы декоратора как информацию состояния для применения в фактическом декораторе, в вызываемом объекте, который он возвращает, или в обоих. Фрагмент кода предохраняет аргумент информации состояния в ссылках из объемлющих областей видимости, но часто используются также и атрибуты класса.

Другими словами, аргументы декораторов нередко подразумевают наличие *трех уровней вызываемых объектов*: вызываемого объекта для приема аргументов декоратора, возвращающего вызываемый объект, который служит в качестве декоратора и возвращает вызываемый объект для обработки обращений к исходной функции или классу. Каждый из трех уровней может быть функцией либо классом и предохранять состояние в форме ссылок из объемлющих областей видимости или атрибутов класса.

Аргументы декораторов могут применяться при предоставлении значений для инициализации атрибутов, сообщений с трассировкой вызовов, имен атрибутов, подлежащих проверке допустимости, и многого другого — сюда входят конфигурационные параметры любого вида для объектов либо их посредников. Конкретные примеры использования аргументов декораторов будут приведены далее в главе.

Декораторы одновременно управляют функциями и классами

Хотя в остатке главы внимание в основном будет сосредоточено на помещении в оболочку будущих обращений к функциям и классам, важно помнить о том, что механизм декорирования является более универсальным — он представляет собой протокол для прогона функций и классов через любой вызываемый объект немедленно после их создания. Как таковое, декорирование также может применяться для вызова произвольной обработки после создания:

```
def decorator(O):
    # Сохранение или дополнение функции или класса O
    return O

@decorator
def F(): ...                # F = decorator(F)

@decorator
class C: ...                # C = decorator(C)
```

До тех пор, пока мы возвращаем исходный декорированный объект вместо посредника, имеется возможность управлять самими функциями и классами, а не только обращениями к ним в более позднее время. Далее в главе мы увидим более реалистичные примеры, в которых данная идея используется для регистрации вызываемых объектов в API-интерфейсе с декорированием и присваиванием атрибутам функций при их создании.

Реализация декораторов функций

Что касается кода, то в остатке главы мы собираемся исследовать рабочие примеры, которые продемонстрируют только что изложенные концепции декораторов. В текущем разделе иллюстрируется работа нескольким декораторов функций, а в следующем — работа декораторов классов. Затем мы закончим более крупными учебными примерами применения декораторов классов и функций — полными реализациями защиты классов и проверки вхождения значений аргументов в заданный диапазон.

Отслеживание вызовов

Для начала давайте возродим пример трассировки вызовов из главы 32. Ниже определяется и применяется декоратор функции, который подсчитывает количество вызовов декорированной функции и для каждого вызова выводит трассировочное сообщение:

```
# Файл decorator1.py

class tracer:
    def __init__(self, func): # При декорировании @: сохранение исходной функции
        self.calls = 0
        self.func = func
    def __call__(self, *args): # При последующих вызовах: запуск исходной функции
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        self.func(*args)
@tracer
def spam(a, b, c):            # spam = tracer(spam)
    print(a + b + c)          # spam помещается в объект декоратора
```

Обратите внимание, что каждая функция, декорированная классом `tracer`, будет создавать новый экземпляр с собственным объектом сохраненной функции и счетчиком вызовов. Также взгляните на использование синтаксиса `*args` для упаковки и распаковки произвольно большого числа переданных аргументов. Такая универсальность делает возможным применение этого декоратора для помещения в оболочку любой функции с любым количеством позиционных аргументов; текущая версия пока еще не работает с ключевыми аргументами или методами уровня класса и не возвращает результатов, но позже в разделе мы устраним указанные недостатки.

Если теперь мы импортируем функцию модуля и протестируем ее в интерактивном сеансе, то получим показанное далее поведение — каждый вызов изначально генерирует трассировочное сообщение, потому что класс декоратора перехватывает вызов. Код работает в Python 2.X и 3.X, как и весь код в главе, если не указано иное (я сделал вывод нейтральным к версиям, а декораторы не требуют классов нового стиля; кроме того, были сокращены некоторые шестнадцатеричные адреса):

```
>>> from decorator1 import spam
>>> spam(1, 2, 3)      # В действительности вызывается объект-оболочка tracer
call 1 to spam
6
>>> spam('a', 'b', 'c')     # Вызывается метод __call__ в классе
call 2 to spam
abc
>>> spam.calls         # Количество вызовов в информации состояния объекта-оболочки
2
>>> spam
<decorator1.tracer object at 0x02D9A730>
```

Во время выполнения класс tracer сохраняет декорированную функцию и перехватывает будущие ее вызовы, чтобы добавить уровень логики, которая подсчитывает вызовы и выводит для каждого сообщение. Обратите внимание, что общее количество вызовов отображается как атрибут декорированной функции. На самом деле в случае декорирования spam является экземпляром класса tracer, что может иметь последствия для программ, предпринимающих проверку типов, но в целом благоприятные (декораторы могли бы копировать атрибут __name__ исходной функции, но такая подделка ограничена и способна привести к путанице).

Для вызовов функция декораторный синтаксис @ может оказаться удобнее, чем модификация каждого вызова для учета дополнительного уровня логики, и он избегает случайного вызова исходной функции напрямую. Вот эквивалентная реализация без декораторов:

```
calls = 0
def tracer(func, *args):
    global calls
    calls += 1
    print('call %s to %s' % (calls, func.__name__))
    func(*args)

def spam(a, b, c):
    print(a, b, c)

>>> spam(1, 2, 3)          # Нормальный вызов без отслеживания: случайный?
1 2 3

>>> tracer(spam, 1, 2, 3) # Специальный вызов с отслеживанием без декораторов
call 1 to spam
1 2 3
```

Такая альтернативная версия может использоваться с любой функцией без специального синтаксиса @, но в отличие от версии с декоратором она требует добавочного синтаксиса в каждом месте, где функция вызывается в коде. Кроме того, ее намерение может быть не настолько очевидным, а гарантия того, что дополнительный уровень будет задействован для нормальных вызовов, попросту отсутствует. Хотя декораторы никогда не считаются *обязательными* (мы всегда можем вручную повторно привязывать имена), они часто являются самым удобным и унифицированным вариантом.

Варианты предохранения состояния для декораторов

В последнем примере предыдущего раздела поднята важная проблема. Декораторы функций предлагают разнообразные варианты предохранения информации состояния, предоставленной во время декорирования, для применения в течение фактического вызова функции. Как правило, они должны поддерживать множество декорированных объектов и множество вызовов, но есть несколько способов достижения таких целей: для предохранения состояния можно использовать атрибуты экземпляров, глобальные переменные, нелокальные переменные замыканий и атрибуты функций.

Атрибуты экземпляров классов

Ниже показана расширенная версия предыдущего примера с добавленной поддержкой *ключевых* аргументов посредством синтаксиса **, которая к тому же *возвращает* результат внутренней функции для охвата большего числа сценариев применения (тем, кто читает книгу непоследовательно, сначала потребуется изучить ключевые аргументы в главе 18 первого тома):

```
class tracer:                         # Состояние через атрибуты экземпляра
   def __init__(self, func):          # При декорировании @
      self.calls = 0                  # Сохранить функцию для вызова в будущем
      self.func = func
   def __call__(self, *args, **kwargs):    # При вызове исходной функции
      self.calls += 1
      print('call %s to %s' % (self.calls, self.func.__name__))
      return self.func(*args, **kwargs)

@tracer
def spam(a, b, c):                    # То же, что и spam = tracer(spam)
   print(a + b + c)                   # Запускается tracer.__init__

@tracer
def eggs(x, y):                       # То же, что и eggs = tracer(eggs)
   print(x ** y)                      # eggs помещается в объект tracer

spam(1, 2, 3)                         # Вызывается экземпляр tracer:
                                      #   запускается tracer.__call__
spam(a=4, b=5, c=6)                   # spam - атрибут экземпляра

eggs(2, 16)          # Вызывается экземпляр tracer, self.func - это eggs
eggs(4, y=4)         # self.calls - значение для каждого случая декорирования
```

Как и в первоначальной версии, для явного хранения состояния здесь используются *атрибуты экземпляров класса*. Внутренняя функция и счетчик вызовов представляют информацию *для каждого экземпляра* — каждый случай декорирования получает собственную копию. При запуске как сценария в Python 2.X или 3.X вывод данной версии будет следующим; обратите внимание на то, что функции spam и eggs имеют свои счетчики вызовов, поскольку каждый случай декорирования создает новый экземпляр класса:

```
c:\code> python decorator2.py
call 1 to spam
6
call 2 to spam
15
call 1 to eggs
65536
call 2 to eggs
256
```

Несмотря на то что такая кодовая схема удобна для декорирования функций, она по-прежнему сопряжена с проблемами, когда применяется к методам — недостаток, который мы устраним позже.

Объемлющие области видимости и глобальные переменные

Функции замыканий — со ссылками из объемлющих областей видимости def и вложенными операторами def — часто могут обеспечить тот же самый эффект, особенно для статических данных вроде декорированной исходной функции. Однако в этом примере нам также понадобится счетчик в объемлющей области видимости, который *изменяется* при каждом вызове, что невозможно в Python 2.X (вспомните из главы 17 первого тома, что оператор nonlocal доступен только в Python 3.X).

В Python 2.X мы можем использовать либо классы и атрибуты, как в предыдущем разделе, либо другие варианты. Одним из кандидатов будет перенос переменных состояния в глобальную область видимости с объявлениями, давая в результате код, который работает в Python 2.X и 3.X:

```
calls = 0
def tracer(func):                    # Состояние через объемлющую область
                                     # видимости и глобальную переменную
    def wrapper(*args, **kwargs):    # Вместо атрибутов класса
        global calls   # calls - глобальная переменная, не для каждой функции
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer
def spam(a, b, c):                   # То же, что и spam = tracer(spam)
    print(a + b + c)

@tracer
def eggs(x, y):                      # То же, что и eggs = tracer(eggs)
    print(x ** y)

spam(1, 2, 3)         # На самом деле вызывается wrapper, присвоенный spam
spam(a=4, b=5, c=6)   # wrapper вызывает spam

eggs(2, 16)           # На самом деле вызывается wrapper, присвоенный eggs
eggs(4, y=4)          # Глобальная переменная calls не является
                      # отдельной для каждого случая декорирования!
```

К сожалению, перемещение счетчика в общую глобальную область видимости, чтобы сделать возможным его изменение, также означает, что он будет *разделяться* всеми внутренними функциями. В отличие от атрибутов экземпляров классов глобальные счетчики относятся ко всей программе, а не к каждой функции — счетчик инкрементируется для вызова *любой* отслеживаемой функции. Вы сможете заметить разницу, если сравните вывод этой версии с выводом предыдущей версии — единственный разделяемый глобальный счетчик вызовов некорректно обновляется обращениями к каждой декорированной функции:

```
c:\code> python decorator3.py
call 1 to spam
6
call 2 to spam
15
call 3 to eggs
65536
call 4 to eggs
256
```

Объемлющие области видимости и нелокальные переменные

В ряде случаев разделяемое глобальное состояние может быть именно тем, что нужно. Тем не менее, если на самом деле вы хотите иметь счетчик *для каждой функции*, тогда можете применять либо классы, как ранее, либо функции *замыканий* (они же *фабричные* функции) и оператор `nonlocal` в Python 3.X, описанный в главе 17 первого тома. Поскольку оператор `nonlocal` разрешает изменять переменные из областей видимости объемлющих функций, они могут выступать в качестве изменяемых данных для каждого случая декорирования, но только в Python 3.X:

```
def tracer(func):          # Состояние через объемлющую область
                           # видимости и нелокальную переменную
    calls = 0              # Вместо атрибутов класса или глобальных переменных
```

```
    def wrapper(*args, **kwargs):       # calls - для каждой функции,
                                        #  не глобальная переменная
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer
def spam(a, b, c):    # То же, что и spam = tracer(spam)
    print(a + b + c)

@tracer
def eggs(x, y):       # То же, что и eggs = tracer(eggs)
    print(x ** y)

spam(1, 2, 3)         # На самом деле вызывается wrapper, привязанный к spam
spam(a=4, b=5, c=6)   # wrapper вызывает spam

eggs(2, 16)           # На самом деле вызывается wrapper, привязанный к eggs
eggs(4, y=4)          # Нелокальная переменная _является_ отдельной
                      # для каждого случая декорирования!
```

Теперь из-за того, что переменные из объемлющей области видимости не являются глобальными переменными, относящимися ко всей программе, каждая внутренняя функция снова получает собственный счетчик, как было при использовании классов и атрибутов. Вот новый вывод, полученный в результате запуска под управлением Python 3.X:

```
c:\code> py -3 decorator4.py
call 1 to spam
6
call 2 to spam
15
call 1 to eggs
65536
call 2 to eggs
256
```

Атрибуты функций

Наконец, даже если вы не работаете с Python 3.X и не располагаете доступом к оператору `nonlocal` или хотите обеспечить переносимость своего кода между линейками Python 3.X и Python 2.X, то все равно имеете возможность избежать использования глобальных переменных и классов, взамен задействовав для определенного изменяемого состояния *атрибуты функций*. Во всех версиях, начиная с Python 2.1, мы можем присоединять к функциям произвольные атрибуты за счет присваивания им значений с помощью выражения вида *функция.атрибут = значение*. Поскольку фабричная функция при любом вызове создает новую функцию, ее атрибуты становятся состоянием для каждого вызова. Кроме того, вам нужно применять такую методику только для переменных состояния, которые должны *изменяться*; ссылки из объемлющих областей видимости по-прежнему предохраняются и нормально работают.

В нашем примере мы можем просто использовать для состояния `wrapper.calls`. Следующая версия работает аналогично предыдущей версии с `nonlocal`, т.к. счетчик снова относится к каждой декорированной функции, но она также выполняется в Python 2.X:

```
def tracer(func):                         # Состояние через объемлющую область
                                          #   видимости и атрибуты функций
    def wrapper(*args, **kwargs):         # calls - для каждой функции,
                                          #   не глобальная переменная
        wrapper.calls += 1
        print('call %s to %s' % (wrapper.calls, func.__name__))
        return func(*args, **kwargs)
    wrapper.calls = 0
    return wrapper

@tracer
def spam(a, b, c):              # То же, что и spam = tracer(spam)
    print(a + b + c)

@tracer
def eggs(x, y):                 # То же, что и eggs = tracer(eggs)
    print(x ** y)

spam(1, 2, 3)                   # На самом деле вызывается wrapper, присвоенный spam
spam(a=4, b=5, c=6)             # wrapper вызывает spam

eggs(2, 16)                     # На самом деле вызывается wrapper, присвоенный eggs
eggs(4, y=4)                    # Значение wrapper.calls _относится_
                                #   к каждому случаю декорирования
```

Как объяснялось в главе 17 первого тома, код работает лишь потому, что имя wrapper сохранено в области видимости объемлющей функции tracer. Когда позже мы инкрементируем wrapper.calls, то не изменяем само имя wrapper, поэтому объявление nonlocal не требуется. Данная версия выполняется в обеих линейках Python:

```
c:\code> py -2 decorator5.py
...такой же вывод, как у предыдущей версии, но работает и в Python 2.X...
```

Представленная выше схема чуть не была переведена в категорию сноски, т.к. она может оказаться даже менее ясной, чем версия с nonlocal в Python 3.X, и поэтому ее лучше приберечь для ситуаций, где другие схемы ничем не помогут. Однако атрибуты функций также обладают существенными преимуществами. С одной стороны, они делают возможным доступ к сохраненному состоянию *снаружи* кода декоратора; нелокальные переменные могут быть видны только внутри самой вложенной функции, но атрибуты функций имеют более широкую видимость. С другой стороны, они гораздо более *переносимы*; схема также работает в Python 2.X, что делает ее нейтральной к версиям.

Атрибуты функций будут снова задействованы при ответе на один из контрольных вопросов главы, где их видимость снаружи вызываемых объектов становится преимуществом. Поскольку изменяемое состояние связано с контекстом применения, атрибуты функций эквивалентны нелокальным переменным из объемлющих областей видимости. Как обычно, выбор среди множества инструментов является неотъемлемой частью задачи программирования.

Из-за того, что декораторы часто подразумевают наличие множества уровней вызываемых объектов, вы можете комбинировать функции с объемлющими областями видимости, классы с атрибутами и атрибуты функций, чтобы добиться многосторонности кодовых структур. Тем не менее, позже вы увидите, что иногда задача оказывается более тонкой, чем ожидалось — каждая декорированная функция должна иметь собственное состояние, а каждый декорированный класс может требовать состояния и для себя, и для каждого создаваемого экземпляра.

В следующем разделе подробно объясняется, что если мы хотим применять декораторы функций также к методам уровня класса, тогда обязаны позаботиться о разграничении, которое Python делает между декораторами, реализованными как вызываемые объекты экземпляров классов, и декораторами, записанными в виде функций.

Грубые ошибки, связанные с классами, часть I: декорирование методов

Когда я занимался реализацией первого основанного на классах декоратора функций `tracer` в файле `decorator1.py`, то наивно полагал, что его также удастся применять к любому *методу*. Я рассуждал, что декорированные методы должны работать аналогично, а добавляемый автоматически аргумент экземпляра `self` просто будет включен в начало `*args`. Единственный реальный недостаток такого предположения заключается в том, что оно *глубоко ошибочно*! В случае применения к методу класса первая версия `tracer` терпит неудачу, т.к. `self` является экземпляром класса декоратора, а экземпляр декорированного целевого класса вообще не помещается в `*args`. Сказанное справедливо и для Python 3.X, и для Python 2.X.

Я представил данное явление ранее в главе, но теперь мы можем взглянуть на него в контексте реалистичного рабочего кода. Для следующего декоратора отслеживания на основе класса:

```
class tracer:
    def __init__(self, func):         # При декорировании @
        self.calls = 0                # Сохранение функции для вызова в будущем
        self.func = func
    def __call__(self, *args, **kwargs):  # При обращении к исходной функции
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
```

декорирование простых функций работает, как было объявлено ранее:

```
@tracer
def spam(a, b, c):                    # spam = tracer(spam)
    print(a + b + c)                  # Запускается tracer.__init__

>>> spam(1, 2, 3)                     # Выполняется tracer.__call__
call 1 to spam
6
>>> spam(a=4, b=5, c=6)               # spam хранится в атрибуте экземпляра
call 2 to spam
15
```

Однако декорирование методов уровня класса потерпит неудачу (более рассудительные последовательные читатели могут признать приведенный далее код как адаптацию нашего класса `Person`, который взят из учебного руководства по ООП, предложенного в главе 28):

```
class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

@tracer
def giveRaise(self, percent):         # giveRaise = tracer(giveRaise)
    self.pay *= (1.0 + percent)
```

```
    @tracer
    def lastName(self):                        # lastName = tracer(lastName)
        return self.name.split()[-1]

>>> bob = Person('Bob Smith', 50000)           # tracer запоминает функции методов
>>> bob.giveRaise(.25)                         # Запускается tracer.__call__(???, .25)
call 1 to giveRaise
TypeError: giveRaise() missing 1 required positional argument: 'percent'
Ошибка типа: в giveRaise() отсутствует 1 обязательный позиционный аргумент:
percent
>>> print(bob.lastName())                      # Запускается tracer.__call__(???)
call 1 to lastName
TypeError: lastName() missing 1 required positional argument: 'self'
Ошибка типа: в lastName() отсутствует 1 обязательный позиционный аргумент: self
```

Корень проблемы здесь кроется в аргументе `self` метода `__call__` класса `tracer` – он является экземпляром `tracer` или же экземпляром `Person`? На самом деле нам необходимы в коде *оба* экземпляра: экземпляр `tracer` для состояния декоратора и экземпляр `Person` для перенаправления на исходный метод. В действительности `self` *обязан* быть объектом `tracer`, чтобы предоставить доступ к информации состояния `tracer` (его атрибутам `calls` и `func`); это справедливо для декорирования как простой функции, так и метода.

К сожалению, когда имя декорированного метода повторно привязывается к объекту экземпляра класса с помощью `__call__`, интерпретатор Python передает в `self` только *экземпляр* `tracer`; он вообще не передает экземпляр `Person` в списке аргументов. Кроме того, поскольку экземпляру `tracer` ничего не известно об экземпляре `Person`, который мы пытаемся обработать посредством вызовов методов, создать связанный метод с экземпляром не удастся, следовательно, нет способа для корректного координирования вызова. Это не ошибка, но очень тонкая особенность.

В итоге предыдущий код передает декорированному методу слишком мало аргументов, что приводит к ошибке. Для проверки добавьте в метод `__call__` декоратора вывод всех его аргументов – вы увидите, что `self` представляет собой экземпляр `tracer`, а экземпляр `Person` отсутствует:

```
>>> bob.giveRaise(.25)
<__main__.tracer object at 0x02A486D8> (0.25,) {}
call 1 to giveRaise
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __call__
TypeError: giveRaise() missing 1 required positional argument: 'percent'
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
  Файл <stdin>, строка 9, в __call__
Ошибка типа: в giveRaise() отсутствует 1 обязательный позиционный аргумент:
percent
```

Как упоминалось ранее, так происходит из-за того, что интерпретатор Python передает подразумеваемый экземпляр аргументу `self`, только когда имя метода привязано к простой функции; если оно является экземпляром вызываемого класса, тогда взамен передается экземпляр этого класса. Формально интерпретатор Python создает объект связанного метода, содержащий экземпляр целевого класса, только когда метод представляет собой простую функцию, а не вызываемый экземпляр еще одного класса.

Использование вложенных функций для декорирования методов

Если вы хотите, чтобы декораторы функций работали с простыми функциями *и* с методами уровня классов, то самое прямолинейное решение предусматривает применение одной из других методик предохранения состояния, которые были описаны ранее. Речь идет о реализации декоратора функций в виде вложенного оператора `def`, чтобы не полагаться на единственный аргумент экземпляра `self`, который будет как экземпляром класса оболочки, так и экземпляром целевого класса.

В показанной ниже альтернативной версии затруднение решается с использованием нелокальной переменной Python 3.X; чтобы код работал в Python 2.X, его необходимо переписать, применяя атрибуты функций для изменяемого состояния `calls`. Поскольку декорированные методы повторно привязываются к простым функциям, а не объектам экземпляром, интерпретатор Python корректно передает объект `Person` в первом аргументе, а декоратор передает его из первого элемента `*args` аргументу `self` действительного декорированного метода:

```
# Декоратор отслеживания вызовов для функций и методов

def tracer(func):                  # Использовать с __call__ функцию, но не класс
    calls = 0                      # Иначе self - только экземпляр декоратора!
    def onCall(*args, **kwargs):   # Или применить [onCall.calls += 1]
                                   #   для Python 2.X+3.X
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall

if __name__ == '__main__':

    # Применяется к простым функциям
    @tracer
    def spam(a, b, c):             # spam = tracer(spam)
        print(a + b + c)           # onCall запоминает spam
    @tracer
    def eggs(N):
        return 2 ** N
    spam(1, 2, 3)                  # Запускается onCall(1, 2, 3)
    spam(a=4, b=5, c=6)
    print(eggs(32))

    # Применяется также к функциям методов уровня класса!
    class Person:
        def __init__(self, name, pay):
            self.name = name
            self.pay = pay
        @tracer
        def giveRaise(self, percent):     # giveRaise = tracer(giveRaise)
            self.pay *= (1.0 + percent)   # onCall запоминает giveRaise

        @tracer
        def lastName(self):               # lastName = tracer(lastName)
            return self.name.split()[-1]
    print('methods...')                   # методы...
    bob = Person('Bob Smith', 50000)
    sue = Person('Sue Jones', 100000)
    print(bob.name, sue.name)
    sue.giveRaise(.10)                        # Запускается onCall(sue, .10)
```

```
    print(int(sue.pay))
    print(bob.lastName(), sue.lastName())   # Запускается onCall(bob),
                                            #   lastName в области видимости
```

Мы также поместили код самотестирования файла внутрь оператора проверки `__name__`, так что декоратор можно импортировать и использовать где-то в другом месте. Эта версия работает одинаково с функциями и методами, но выполняется только под управлением Python 3.X из-за применения nonlocal:

```
c:\code> py -3 calltracer.py
call 1 to spam
6
call 2 to spam
15
call 1 to eggs
4294967296
methods...
Bob Smith Sue Jones
call 1 to giveRaise
110000
call 1 to lastName
call 2 to lastName
Smith Jones
```

Отследите приведенные результаты, чтобы лучше понимать модель; в следующем разделе предлагается альтернативная версия, которая поддерживает классы, но также характеризуется гораздо большей сложностью.

Использование дескрипторов для декорирования методов

Несмотря на то что решение с вложенными функциями, продемонстрированное в предыдущем разделе, является наиболее прямолинейным способом поддержки декораторов, которые применяются к функциям и методам уровня класса, возможны другие схемы. Скажем, здесь также может помочь средство *дескрипторов*, исследованное в предыдущей главе.

Вспомните из нашего обсуждения в той главе, что дескриптор обычно представляет собой атрибут класса, которому присваивается объект с методом `__get__`, автоматически запускаемым всякий раз, когда производится ссылка и извлечение атрибута. Наследование от object, принятое в классах нового стиля, требуется для дескрипторов в Python 2.X, но не в Python 3.X:

```
class Descriptor(object):
    def __get__(self, instance, owner): ...

class Subject:
    attr = Descriptor()

X = Subject()
X.attr   # Грубо говоря, запускает Descriptor.__get__(Subject.attr, X, Subject)
```

Дескрипторы также могут иметь методы доступа `__set__` и `__del__`, но здесь они не нужны. В настоящей главе более важен метод `__get__` дескриптора, поскольку во время вызова он принимает экземпляры класса дескриптора *и* целевого класса, что хорошо подходит для декорирования методов, когда при координировании вызовов нам необходимо как состояние декоратора, так и экземпляр исходного класса. Рассмотрим следующую альтернативную версию декоратора отслеживания, который *также* оказывается дескриптором в случае использования для метода уровня класса:

```
class tracer(object):                # Декоратор + дескриптор
    def __init__(self, func):        # При декорировании @
        self.calls = 0               # Сохранение функции для вызова в будущем
        self.func = func
    def __call__(self, *args, **kwargs):  # При обращении к исходной функции
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):   # При извлечении атрибутов методов
        return wrapper(self, instance)

class wrapper:
    def __init__(self, desc, subj):       # Сохранение обоих экземпляров
       self.desc = desc           # Направление вызовов декоратору/дескриптору
       self.subj = subj
    def __call__(self, *args, **kwargs):
       return self.desc(self.subj, *args, **kwargs) # Запускается tracer.__call__

@tracer
def spam(a, b, c):                           # spam = tracer(spam)
    ...то же код, что и раньше...            # Использует только __call__

class Person:
    @tracer
    def giveRaise(self, percent):            # giveRaise = tracer(giveRaise)
        ...то же код, что и раньше...        # Делает giveRaise дескриптором
```

Текущая версия декоратора `tracer` работает аналогично предшествующей версии с вложенными функциями, но действие варьируется в зависимости от контекста применения.

- Декорированные функции вызывают только его метод `__call__` и никогда его метод `__get__`.
- Декорированные методы вызывают сначала его метод `__get__`, чтобы выполнить извлечение имени метода (для `I.метод`); возвращенный методом `__get__` объект хранит экземпляр целевого класса и затем вызывается с целью завершения выражения вызова, тем самым запуская метод `__call__` декоратора (для `()`).

Например, вызов в тестовом коде:

```
sue.giveRaise(.10)    # Запускается __get__, затем __call__
```

выполняет сначала `tracer.__get__`, потому что атрибут `giveRaise` в классе `Person` был повторно привязан к дескриптору декоратором функций методов. Затем выражение вызова запускает метод `__call__` возвращенного объекта `wrapper`, который в свою очередь вызывает `tracer.__call__`. Другими словами, вызовы декорированных методов инициируют процесс из четырех шагов: `tracer.__get__`, за которым следуют три операции вызова — `wrapper.__call__`, `tracer.__call__` и в заключение исходного декорированного метода.

Объект `wrapper` хранит экземпляры дескриптора и целевого класса, так что он может направить управление исходному экземпляру класса декоратора/дескриптора. В действительности объект `wrapper` сохраняет экземпляр целевого класса, доступный во время извлечения атрибута метода, и добавляет его в список аргументов более позднего вызова, который передается методу `__call__` декоратора. В этом приложении требуется направление вызова обратно экземпляру класса дескриптора, чтобы все обращения к внутреннему методу использовали ту же самую информацию состояния счетчика `calls` в объекте экземпляра дескриптора.

В качестве альтернативы для обеспечения того же эффекта мы могли бы применить вложенную функцию или ссылки из объемлющих областей видимости — следующая версия работает так же, как предыдущая, за счет обмена местами атрибутов класса и объекта для вложенной функции и ссылок. Она требует значительно меньше кода, но при каждом вызове декорированного метода следует такому же процессу из четырех шагов:

```
class tracer(object):
    def __init__(self, func):          # При декорировании @
        self.calls = 0                 # Сохранение функции для вызова в будущем
        self.func = func
    def __call__(self, *args, **kwargs):  # При обращении к исходной функции
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):       # При извлечении метода
        def wrapper(*args, **kwargs):         # Сохранение обоих экземпляров
            return self(instance, *args, **kwargs)    # Запускается __call__
        return wrapper
```

Добавьте операторы `print` к методам альтернативных версий, чтобы самостоятельно отследить многошаговый процесс извлечения/вызова, и запустите их с тем же тестовым кодом, как ранее делалось для версии с вложенными функциями (исходный код находится в файле `calltracer-descr.py`). В любой из двух версий основанная на дескрипторах схема также гораздо хитроумнее, чем вариант с вложенными функциями, и вероятно потому рассматривается здесь во вторую очередь. Говоря более прямо, если сложность данной версии не заставляет вас мучиться от кошмаров по ночам, то издержки в плане производительности наверняка должны! Тем не менее, в других контекстах такая кодовая схема может оказаться полезной.

Стоит также отметить, что мы можем реализовать декоратор на основе дескриптора проще, как показано ниже, но тогда он был бы применим только к методам, а не к простым функциям. Это внутренне присущее ограничение дескрипторов атрибутов (и всего лишь противоположность задачи, которую мы пытаемся решить: приложение к функциям и методам):

```
class tracer(object):                        # Для методов, но не для функций!
    def __init__(self, meth):                # При декорировании @
        self.calls = 0
        self.meth = meth
    def __get__(self, instance, owner):      # При извлечении метода
        def wrapper(*args, **kwargs):        # При вызове метода: посредник
                                             #   с self и экземпляром
            self.calls += 1
            print('call %s to %s' % (self.calls, self.meth.__name__))
            return self.meth(instance, *args, **kwargs)
        return wrapper

class Person:
    @tracer                                  # Применяется к методам класса
    def giveRaise(self, percent):            # giveRaise = tracer(giveRaise)
        ...                                  # Делает giveRaise дескриптором

    @tracer                 # Но терпит неудачу с простыми функциями
    def spam(a, b, c):      # spam = tracer(spam)
        ...                 # Здесь никаких извлечений атрибутов не происходит
```

В остатке данной главы мы обираемся довольно бессистемно относиться к использованию классов или функций при реализации декораторов функций, пока они применяются только к функциям. Некоторые декораторы могут не требовать экземпляра исходного класса, и по-прежнему будут работать с функциями и методами, если они реализованы в виде классов. Что-нибудь наподобие собственного декоратора `staticmethod` в Python, например, не требует экземпляра целевого класса (более того, весь его смысл связан с тем, чтобы устранить экземпляр из вызова).

Однако мораль этой истории в том, что если вы хотите, чтобы декораторы работали с простыми функциями и методами, то вероятно лучше задействовать обрисованную здесь кодовую схему на основе вложенных функций, а не класс с перехватом вызовов.

Измерение времени вызовов

Для получения более полного представления о том, на что способны декораторы функций, давайте возьмем другой сценарий использования. Наш следующий декоратор измеряет время вызовов декорированной функции — время одного вызова и суммарное время всех вызовов. Декоратор применяется к двум функциям с целью сравнения относительной скорости, с которой выполняются списковые включения и встроенный вызов `map`:

```
# Файл timerdeco1.py
# Предостережение: область все же отличается - список в Python 2.X,
#  итерируемый объект в Python 3.X
# Предостережение: в таком виде timer не будет работать с методами
#  (см. ответ на контрольный вопрос)
import time, sys
force = list if sys.version_info[0] == 3 else (lambda X: X)

class timer:
    def __init__(self, func):
        self.func = func
        self.alltime = 0
    def __call__(self, *args, **kargs):
        start = time.clock()
        result = self.func(*args, **kargs)
        elapsed = time.clock() - start
        self.alltime += elapsed
        print('%s: %.5f, %.5f' % (self.func.__name__, elapsed, self.alltime))
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return force(map((lambda x: x * 2), range(N)))

result = listcomp(5)          # Время для этого вызова, всех вызовов
                              #  и возвращаемое значение
listcomp(50000)
listcomp(500000)
listcomp(1000000)
print(result)
print('allTime = %s' % listcomp.alltime)    # Суммарное время для всех
                                            #  вызовов listcomp
```

```
print('')
result = mapcall(5)
mapcall(50000)
mapcall(500000)
mapcall(1000000)
print(result)
print('allTime = %s' % mapcall.alltime)      # Суммарное время для всех
                                             #   вызовов mapcall

print('\n**map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))
```

При запуске в Python 3.X или 2.X вывод кода самотестирования данного файла выглядит так, как показано ниже. Для каждого вызова он содержит имя функции, время выполнения этого вызова и время выполнения всех вызовов, совершенных до сих пор. Кроме того, указывается возвращаемое значение первого вызова, совокупное время выполнения каждой функции и в конце соотношение времени выполнения списковых включений и встроенного вызова map:

```
c:\code> py -3 timerdeco1.py
listcomp: 0.00001, 0.00001
listcomp: 0.00499, 0.00499
listcomp: 0.05716, 0.06215
listcomp: 0.11565, 0.17781
[0, 2, 4, 6, 8]
allTime = 0.17780527629411225

mapcall: 0.00002, 0.00002
mapcall: 0.00988, 0.00990
mapcall: 0.10601, 0.11591
mapcall: 0.21690, 0.33281
[0, 2, 4, 6, 8]
allTime = 0.3328064956447921

**map/comp = 1.872
```

Конечно, показатели времени варьируются в зависимости от линейки Python и компьютера, используемого для тестирования, а совокупное время доступно в виде атрибута экземпляра класса. Как обычно, вызовы map почти в два раза медленнее списковых включений, когда последним удается избежать вызова функции (или, что эквивалентно, потребность в вызовах функций может сделать map медленнее).

Декораторы или измерение времени для каждого вызова

Для сравнения возьмем представленный в главе 21 первого тома подход *без декораторов* к измерению времени выполнения итерационных альтернатив. Там мы оценивали две методики измерения времени каждого вызова собственной и библиотечной реализаций. Здесь мы вводим в действие измерение времени для случая тестового кода декоратора с 1 миллионом списковых включений, хотя и с дополнительными затратами со стороны управляющего кода, включая внешний цикл и вызовы функций:

```
>>> def listcomp(N): [x * 2 for x in range(N)]

>>> import timer                          # Методики из главы 21 первого тома
>>> timer.total(1, listcomp, 1000000)
(0.1461295268088542, None)

>>> import timeit
>>> timeit.timeit(number=1, stmt=lambda: listcomp(1000000))
0.14964829430189397
```

В этом конкретном случае подход, в котором декораторы не применяются, позволил бы использовать целевые функции с измерением времени или без него, но также бы усложнил сигнатуру вызовов, когда измерение времени желательно – нам пришлось бы добавлять код к каждому вызову, а не один раз к def. Более того, в схеме без декораторов отсутствовал бы прямой способ гарантировать, что все вызовы построителя списка в программе прогоняются через логику таймера, кроме поиска и возможного изменения их всех. В итоге могли бы возникнуть затруднения со сбором накопленных данных для всех вызовов.

В целом *декораторы* могут оказаться предпочтительным вариантом, когда функции уже введены в действие как часть более крупной системы, а потому их вызовы нелегко передавать функциям анализа. С другой стороны, поскольку декораторы нагружают каждый вызов дополняющей логикой, подход *без декораторов* может быть эффективнее, если вы хотите дополнять вызовы более избирательно. Как обычно, разные инструменты исполняют разные роли.

Переносимость вызовов таймера и новые варианты, которые появились в версии Python 3.3. Просмотрите еще раз более полную обработку и выбор функций модуля time в главе 21 первого тома плюс врезку "Новые вызовы таймеров, появившиеся в версии Python 3.3" в той же главе, где обсуждаются новые и усовершенствованные функции таймера в данном модуле, которые стали доступными, начиная с версии Python 3.3 (скажем, perf_counter). Здесь мы принимаем упрощенный подход ради краткости и нейтральности к версиям, но time.clock может быть не наилучшим выбором на ряде платформ даже до выхода Python 3.3, и вне среды Windows могут требоваться проверки на предмет платформы и версии.

Тонкости тестирования

Обратите внимание на то, что в сценарии timerdeco1.py применяется force для обеспечения его переносимости между линейками Python 2.X и Python 3.X. Как было описано в главе 14 первого тома, встроенная функция map возвращает *итерируемый объект*, который генерирует результаты по запросу в Python 3.X, но действительный список в Python 2.X. Следовательно, сама по себе функция map из Python 3.X не сравнивается напрямую с работой списковых включений. Фактически, если вызов map не помещен внутрь вызова list для принудительного выпуска результатов, то тест map практически вообще не отнимает какое-то время в Python 3.X – функция map возвращает итерируемый объект без итерирования!

В то же время добавление вызова list также и в Python 2.X налагает на map несправедливый штраф – результаты теста map будут включать время, требуемое для построения *двух* списков, а не одного. Чтобы обойти проблему, сценарий выбирает объемлющую функцию map согласно номеру версии Python в sys: для Python 3.X выбирается list, а для Python 2.X используется пустая функция, которая просто возвращает свой входной аргумент без изменений. Это добавляет весьма несущественное постоянное время в Python 2.X, которое вероятно окажется полностью перекрытым затратами на итерации внутреннего цикла в хронометрируемой функции.

Наряду с тем, что такой прием делает сравнение списковых включений и встроенной функции map более справедливым в любой линейке Python 2.X или Python 3.X, поскольку range также является итератором в Python 3.X, результаты для Python 2.X и 3.X не могут сравниваться напрямую, если только не изъять данный вызов из хронометрируемого кода. Они относительно сопоставимы – и в любом случае будут отражать рекомендуемый код в каждой линейке, – но итерация range в Python 3.X

добавляет дополнительное время. Сведения подобного рода были представлены при воссоздании эталонных тестов в главе 21 первого тома; производство сопоставимых чисел часто оказывается нетривиальной задачей.

Наконец, как мы ранее поступали для декоратора отслеживания, декоратор измерения времени можно было бы сделать многократно используемым в других модулях, поместив код самотестирования в конец файла внутрь проверки `__name__`, чтобы он выполнялся только при запуске файла, но не при его импортировании. Тем не менее, этим здесь мы заниматься не будем, т.к. собираемся добавить в код еще одну возможность.

Добавление аргументов к декоратору

Декоратор `timer` из предыдущего раздела работоспособен, но было бы неплохо, если бы он стал более конфигурируемым – например, в универсальном инструменте подобного рода может быть полезной возможность предоставления выходной метки и включения/отключения трассировочных сообщений. Здесь пригодятся аргументы декоратора: при надлежащей реализации мы можем применять их для указания конфигурационных параметров, которые допускают варьирование в зависимости от декорированной функции. Скажем, вот как можно было бы добавлять метку:

```
def timer(label=''):
    def decorator(func):
        def onCall(*args):        # Многоуровневое предохранение состояния:
            ...                   # аргументы передаются функции
            func(*args)           # func хранится в объемлющей области видимости
            print(label, ...      # label хранится в объемлющей области видимости
        return onCall
    return decorator              # Возвращается действительный декоратор

@timer('==>')                     # Подобно listcomp = timer('==>')(listcomp)
def listcomp(N): ...              # listcomp повторно привязывается
                                  #   к новой функции onCall

listcomp(...)                     # В действительности вызывается onCall
```

В коде добавляется объемлющая область видимости, чтобы предохранить аргумент декоратора для использования в будущем вызове. Когда функция `listcomp` определена, интерпретатор Python на самом деле вызывает `decorator` – результат выполнения `timer` перед тем, как фактически происходит декорирование, – со значением `label`, доступным в объемлющей области видимости декоратора. То есть `timer` *возвращает* декоратор, запоминающий аргумент декоратора и исходную функцию, а также возвращающий вызываемый объект `onCall`, который в конечном итоге обращается к исходной функции при вызовах в более позднее время. Поскольку такая структура создает новые функции `decorator` и `onCall`, их объемлющие области видимости сохраняют состояние для каждого случая декорирования.

Мы можем задействовать данную структуру в нашем таймере, чтобы сделать возможной передачу метки и флага управления трассировочными сообщениями на стадии декорирования. Ниже приведен пример такой реализации, помещенный в файл модуля по имени `timerdeco2.py`, который можно импортировать как универсальный инструмент; для организации второго уровня предохранения состояния вместо вложенной функции в нем применяется класс, но совокупный результат аналогичен:

```
import time
def timer(label='', trace=True):          # При аргументах декоратора:
                                          #  сохранение аргументов
    class Timer:
        def __init__(self, func):         # При декорировании @: сохранение
                                          #  декорированной функции
            self.func = func
            self.alltime = 0
        def __call__(self, *args, **kargs): # При вызовах: вызов исходной функции
            start = time.clock()
            result = self.func(*args, **kargs)
            elapsed = time.clock() - start
            self.alltime += elapsed
            if trace:
                format = '%s %s: %.5f, %.5f'
                values = (label, self.func.__name__, elapsed, self.alltime)
                print(format % values)
            return result
    return Timer
```

Почти все, что мы здесь сделали, связано с помещением исходного класса `Timer` внутрь объемлющей функции с целью создания области видимости, которая предохраняет аргументы декоратора для каждого ввода в действие. Внешняя функция `timer` вызывается перед тем, как происходит декорирование, и она просто возвращает класс `Timer`, предназначенный для того, чтобы служить фактическим декоратором. При декорировании создается экземпляр `Timer`, который запоминает саму декорированную функцию, но также имеет доступ к аргументам декоратора в области видимости объемлющей функции.

Измерение времени с аргументами декоратора

На этот раз вместо помещения кода самотестирования в файл `timerdeco2.py` мы будем запускать декоратор в другом файле. Вот клиент нашего декоратора таймера, находящийся в файле модуля `testseqs.py`, который снова применяет его к итерационным альтернативам:

```
import sys
from timerdeco2 import timer
force = list if sys.version_info[0] == 3 else (lambda X: X)

@timer(label='[CCC]==>')
def listcomp(N):                      # Подобно listcomp = timer(...)(listcomp)
    return [x * 2 for x in range(N)] # listcomp(...) triggers Timer.__call__

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return force(map((lambda x: x * 2), range(N)))

for func in (listcomp, mapcall):
    result = func(5)                  # Время для этого вызова, всех вызовов
                                      #  и возвращаемое значение
    func(50000)
    func(500000)
    func(1000000)
    print(result)
    print('allTime = %s\n' % func.alltime) # Суммарное время для всех вызовов

print('**map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))
```

Опять-таки, чтобы сделать сравнение справедливым, в случае Python 3.X обращение к map помещается внутрь вызова list. При запуске в Python 3.X или 2.X файл выводит следующие результаты – каждая декорированная функция теперь имеет собственную метку, определяемую с помощью аргументов декоратора, которая будет более полезной, когда нам необходимо отыскать трассировочные сообщения в выводе крупной программы:

```
c:\code> py -3 testseqs.py
[CCC]==> listcomp: 0.00001, 0.00001
[CCC]==> listcomp: 0.00504, 0.00505
[CCC]==> listcomp: 0.05839, 0.06344
[CCC]==> listcomp: 0.12001, 0.18344
[0, 2, 4, 6, 8]
allTime = 0.1834406801777564

[MMM]==> mapcall: 0.00003, 0.00003
[MMM]==> mapcall: 0.00961, 0.00964
[MMM]==> mapcall: 0.10929, 0.11892
[MMM]==> mapcall: 0.22143, 0.34035
[0, 2, 4, 6, 8]
allTime = 0.3403542519173618

**map/comp = 1.855
```

Как обычно, мы также можем тестировать в интерактивном сеансе, чтобы посмотреть на конфигурационные аргументы декоратора в работе:

```
>>> from timerdeco2 import timer
>>> @timer(trace=False)                    # Без трассировочных сообщений,
                                           #   накопление суммарного времени
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> listcomp.alltime
0.0037191417530599152
>>> listcomp
<timerdeco2.timer.<locals>.Timer object at 0x02957518>

>>> @timer(trace=True, label='\t=>')       # Включение трассировочных сообщений,
                                           #   специальная метка
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
        => listcomp: 0.00106, 0.00106
>>> x = listcomp(5000)
        => listcomp: 0.00108, 0.00214
>>> x = listcomp(5000)
        => listcomp: 0.00107, 0.00321
>>> listcomp.alltime
0.003208920466562404
```

В текущем виде декоратор функций для измерения времени может использоваться с любой функцией, как в модулях, так и в интерактивном сеансе. Другими словами, он автоматически квалифицируется как *универсальный инструмент* для измерения време-

ни выполнения кода в наших сценариях. Дополнительные примеры применения аргументов декораторов ищите в разделах "Реализация закрытых атрибутов" и "Базовый декоратор проверки вхождения значений в диапазон для позиционных аргументов" далее в главе.

Поддержка методов. Рассматриваемый выше декоратор таймера работает с любой *функцией*, но чтобы его можно было применять также к *методам* уровня класса, требуется лишь небольшая переделка. Как иллюстрировалось в разделе "Грубые ошибки, связанные с классами, часть I: декорирование методов" ранее в главе, необходимо избегать использования вложенного класса. Однако поскольку такое изменение умышленно оставлено в качестве одного из контрольных вопросов главы, здесь я воздержусь давать полный ответ.

Реализация декораторов классов

До сих пор мы занимались реализацией декораторов функций для управления обращениями к функциям, но, как было показано, в версиях Python 2.6 и 3.0 декораторы были расширены, чтобы работать также с классами. Наряду с тем, что декораторы классов концептуально похожи на декораторы функций, взамен они применяются к классам — либо для управления самим *классами*, либо для перехвата вызовов, создающих экземпляры, с целью управления *экземплярами*. Также подобно декораторам функций декораторы классов в действительности являются лишь необязательным синтаксическим сахаром, хотя многие считают, что они делают намерение программиста более очевидным и сводят к минимуму ошибочные вызовы или случайный их пропуск.

Классы-одиночки

Из-за того, что декораторы классов способны перехватывать вызовы, создающие экземпляры, они могут использоваться либо для управления всеми экземплярами класса, либо для дополнения интерфейсов этих экземпляров. В целях демонстрации далее приведен первый пример декоратора классов, который решает задачу управления всеми экземплярами класса. В коде реализован паттерн проектирования "Одиночка", при котором в любой момент времени существует самое большее один экземпляр класса. Функция `singleton` определяет и возвращает функцию для управления экземплярами, а посредством синтаксиса `@` целевой класс помещается в оболочку данной функции:

```
# Python 3.X и 2.X: глобальная таблица
instances = {}

def singleton(aClass):                      # При декорировании @
    def onCall(*args, **kwargs):            # При создании экземпляров
        if aClass not in instances:         # Один элемент словаря на класс
            instances[aClass] = aClass(*args, **kwargs)
        return instances[aClass]
    return onCall
```

Чтобы обеспечить применение модели с единственным экземпляром понадобится декорировать желаемые классы (весь рассматриваемый в разделе код находится в файле `singletons.py`):

```
@singleton                                  # Person = singleton(Person)
class Person:                               # Повторная привязка Person к onCall
   def __init__(self, name, hours, rate):   # onCall запоминает Person
      self.name = name
      self.hours = hours
      self.rate = rate
   def pay(self):
      return self.hours * self.rate

@singleton                                  # Spam = singleton(Spam)
class Spam:                                 # Повторная привязка Spam к onCall
   def __init__(self, val):                 # onCall запоминает Spam
      self.attr = val

bob = Person('Bob', 40, 10)                 # В действительности вызывается onCall
print(bob.name, bob.pay())

sue = Person('Sue', 50, 20)                 # То же самое, единственный объект
print(sue.name, sue.pay())

X = Spam(val=42)                            # Один экземпляр Person,
                                            #   один экземпляр Spam
Y = Spam(99)
print(X.attr, Y.attr)
```

Теперь, когда класс Person или Spam используется в будущем с целью создания экземпляра, предоставленный декоратором уровень логики оболочки направляет вызовы, создающие экземпляры, функции onCall, которая гарантирует наличие единственного экземпляра для каждого класса независимо от того, сколько таких вызовов было сделано. Вот вывод кода (Python 2.X добавляет круглые скобки для кортежа):

```
c:\code> python singletons.py
Bob 400
Bob 400
42 42
```

Реализация альтернативных версий

Интересно отметить, что вы можете реализовать более самодостаточное решение, если имеете возможность применять оператор nonlocal (доступный только в Python 3.X) для изменения имен в объемлющей области видимости, как было описано ранее. В показанной ниже альтернативной версии достигается идентичный эффект за счет использования для каждого класса одной *объемлющей области видимости* взамен одной записи в глобальной таблице. Версия работает точно так же, но не полагается на имена в глобальной области видимости вне декоратора (обратите внимание, что здесь при проверке на равенство None можно было бы применять is вместо ==, но в любом случае проверка тривиальна):

```
# Только в Python 3.X: оператор nonlocal

def singleton(aClass):                      # При декорировании @
   instance = None
   def onCall(*args, **kwargs):             # При создании экземпляров
      nonlocal instance                     # Оператор nonlocal в Python 3.X
      if instance == None:
         instance = aClass(*args, **kwargs)       # Одна объемлющая область
                                                  #   видимости на класс
      return instance
   return onCall
```

В Python 3.X или 2.X (Python 2.6 и последующих версиях) вы также можете реализовать самодостаточное решение с помощью либо атрибутов функции, либо класса. Вариант с атрибутами функции воплощен в первой части представленного далее кода, где задействован тот факт, что будет существовать одна *функция* onCall для каждого случая декорирования — пространство имен объекта исполняет такую же роль, как объемлющая область видимости. Во второй части кода используется один *экземпляр* для каждого случая декорирования вместо объемлющей области видимости, объекта функции или глобальной таблицы. На самом деле вторая часть полагается на ту же самую кодовую схему, которую мы позже увидим как распространенную грубую ошибку с классами декораторов — здесь мы *хотим* иметь только один экземпляр, но обычно так не происходит:

```
# Python 3.X и 2.X: атрибуты функций, классы (альтернативные версии)

def singleton(aClass):                          # При декорировании @
    def onCall(*args, **kwargs):                # При создании экземпляров
        if onCall.instance == None:
            onCall.instance = aClass(*args, **kwargs)   # Одна функция на класс
        return onCall.instance
    onCall.instance = None
    return onCall

class singleton:
    def __init__(self, aClass):                 # При декорировании @
        self.aClass = aClass
        self.instance = None
    def __call__(self, *args, **kwargs):        # При создании экземпляров
        if self.instance == None:
            self.instance = self.aClass(*args, **kwargs)    # Один экземпляр
                                                            #   на класс
        return self.instance
```

Чтобы превратить этот декоратор в полноценный универсальный инструмент, нужно сохранить его в импортируемом файле модуля и поместить код самотестирования внутрь проверки `__name__` — шаги, которые мы оставили в качестве упражнения для самостоятельной проработки. Финальная версия, основанная на классе, обеспечивает переносимость и дополнительную структуру, которая может лучше поддерживать будущее развитие, но применение ООП во всех контекстах может оказаться неоправданным.

Отслеживание объектных интерфейсов

Пример с классом-одиночкой в предыдущем разделе иллюстрировал использование декораторов классов для управления *всеми* экземплярами класса. Еще один распространенный сценарий применения для декораторов классов предусматривает дополнение интерфейса *каждого* созданного экземпляра. Декораторы классов могут по существу устанавливать для экземпляров уровень логики оболочки или "посредника", который каким-то образом управляет доступом к их интерфейсам.

Скажем, в главе 31 был показан метод перегрузки операций `__getattr__` как способ оборачивания полных объектных интерфейсов внедренных экземпляров с целью реализации кодовой схемы *делегирования*. Похожие примеры встречались во время раскрытия темы управляемых атрибутов в предыдущей главе. Вспомните, что метод `__getattr__` запускается при извлечении неопределенного имени атрибута; мы можем использовать такую привязку для перехвата обращений к методам в классе контроллера и их передачи внедренному объекту.

Для справки вот исходный пример делегирования без декораторов, который работает с двумя объектами встроенных типов:

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object              # Сохранение объекта
    def __getattr__(self, attrname):
        print('Trace:', attrname)          # Отслеживание извлечения
        return getattr(self.wrapped, attrname)    # Делегирование извлечения

>>> x = Wrapper([1,2,3])                          # Оборачивание списка
>>> x.append(4)                        # Делегирование списковому методу
Trace: append
>>> x.wrapped                          # Вывод элементов
[1, 2, 3, 4]

>>> x = Wrapper({"a": 1, "b": 2})      # Оборачивание словаря
>>> list(x.keys())                     # Делегирование словарному методу
Trace: keys                            # Использовать list() в Python 3.X
['a', 'b']
```

В этом коде класс `Wrapper` перехватывает доступ к любым именованным атрибутам внутреннего объекта, выводит трассировочное сообщение и применяет встроенную функцию `getattr` для передачи запроса внутреннему объекту. В частности, он отслеживает доступ к атрибутам, осуществляемый *снаружи* класса внутреннего объекта; операции доступа внутри методов внутреннего объекта не перехватываются и выполняются нормально по определению. Поведение такой модели *полного интерфейса* отличается от поведения декораторов функций, которые оборачивают только один конкретный метод.

Отслеживание интерфейсов с помощью декораторов классов

Декораторы классов предоставляют альтернативный и удобный способ реализации этой методики с `__getattr__` для оборачивания целого интерфейса. Например, начиная с версий Python 2.6 и 3.0, предыдущий пример класса можно реализовать в виде декоратора класса, который запускает операцию создания экземпляра внутреннего класса вместо передачи готового экземпляра конструктору класса-оболочки. Он также расширен для поддержки ключевых аргументов через `**kargs` и подсчета количества операций доступа, сделанных в целях иллюстрации изменяемого состояния:

```
def Tracer(aClass):                              # При декорировании @
    class Wrapper:
        def __init__(self, *args, **kargs):   # При создании экземпляров
            self.fetches = 0
            self.wrapped = aClass(*args, **kargs) # Использование имени из
                                                  # объемлющей области видимости
        def __getattr__(self, attrname):
            print('Trace: ' + attrname) # Перехват всех атрибутов кроме собственных
            self.fetches += 1
            return getattr(self.wrapped, attrname) # Делегирование
                                                   #   внутреннему объекту
    return Wrapper

if __name__ == '__main__':
    @Tracer
    class Spam:                                    # Spam = Tracer(Spam)
        def display(self):                   # Spam повторно привязывается к Wrapper
            print('Spam!' * 8)
```

```
    @Tracer
    class Person:                                   # Person = Tracer(Person)
        def __init__(self, name, hours, rate): # Wrapper запоминает Person
            self.name = name
            self.hours = hours
            self.rate = rate
        def pay(self):          # Операции доступа извне класса отслеживаются
            return self.hours * self.rate     # Операции доступа внутри
                                            #   методов не отслеживаются

food = Spam()                               # Запускается Wrapper()
food.display()                              # Запускается __getattr__
print([food.fetches])

bob = Person('Bob', 40, 50)     # bob - на самом деле экземпляр Wrapper
print(bob.name)                 # Wrapper содержит внедренный экземпляр Person
print(bob.pay())

print('')
sue = Person('Sue', rate=100, hours=60)     # sue - другой экземпляр Wrapper
print(sue.name)                             # с другим экземпляром Person
print(sue.pay())

print(bob.name)                             # bob имеет отличающееся состояние
print(bob.pay())
print([bob.fetches, sue.fetches])           # Атрибуты Wrapper не отслеживаются
```

Важно отметить, что реализация сильно отличается от декоратора отслеживания, который встречался ранее (вопреки одинаковым именам!). В разделе "Реализация декораторов функций" выше в главе мы взглянули на декораторы, которые позволяют отслеживать и измерять время выполнения вызовов заданной функции или метода. Напротив, за счет перехвата вызовов, создающих экземпляры, декоратор класса здесь дает возможность отслеживать целый объектный интерфейс, т.е. доступ к любым атрибутам экземпляра.

Ниже приведен вывод, выдаваемый кодом в Python 3.X и 2.X (в Python 2.6 и последующих версиях). Операции извлечения атрибутов из экземпляров классов `Spam` и `Person` вызывают логику `__getattr__` в классе `Wrapper`, потому что `food` и `bob` на самом деле являются экземплярами `Wrapper` благодаря перенаправлению вызовов, создающих экземпляры:

```
c:\code> python interfacetracer.py
Trace: display
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
[1]
Trace: name
Bob
Trace: pay
2000

Trace: name
Sue
Trace: pay
6000
Trace: name
Bob
Trace: pay
2000
[4, 2]
```

Обратите внимание на то, что есть один класс Wrapper с предохранением состояния для каждого случая декорирования, созданный вложенным оператором class в функции Tracer, а также на то, что каждый экземпляр получает собственный счетчик операций извлечения в силу генерирования нового экземпляра Wrapper. Как мы увидим позже, организовать это сложнее, чем можно было ожидать.

Применение декораторов классов к встроенным типам

Также важно отметить, что выше декорировался определяемый пользователем класс. В точности как в первоначальном примере из главы 31, мы можем использовать декоратор для помещения в оболочку встроенного типа вроде списка. Условие заключается в том, что мы либо создаем подкласс, чтобы разрешить синтаксис декорирования, либо выполняем декорирование вручную — декораторный синтаксис требует оператора class для строки @. В следующем интерактивном сеансе x снова является экземпляром Wrapper из-за косвенности декорирования:

```
>>> from interfacetracer import Tracer

>>> @Tracer
... class MyList(list): pass    # MyList = Tracer(MyList)

>>> x = MyList([1, 2, 3])       # Запускается Wrapper()
>>> x.append(4)                 # Запускаются __getattr__, append
Trace: append
>>> x.wrapped
[1, 2, 3, 4]

>>> WrapList = Tracer(list)     # Или выполнить декорирование вручную
>>> x = WrapList([4, 5, 6])     # Иначе требуется оператор для создания подкласса
>>> x.append(7)
Trace: append
>>> x.wrapped
[4, 5, 6, 7]
```

Подход с декораторами дает возможность перенести создание экземпляров внутрь самого декоратора, не требуя взамен передачи готового объекта. Хотя отличие может казаться незначительным, оно позволяет сохранить нормальный синтаксис создания экземпляров и претворить в жизнь все преимущества декораторов в целом. Вместо того чтобы требовать прогона всех вызовов, создающих экземпляры, через объект-оболочку вручную, нам понадобится лишь дополнить определения классов декораторным синтаксисом:

```
@Tracer                          # Подход с декораторами
class Person: ...
bob = Person('Bob', 40, 50)
sue = Person('Sue', rate=100, hours=60)

class Person: ...                # Подход без декораторов
bob = Wrapper(Person('Bob', 40, 50))
sue = Wrapper(Person('Sue', rate=100, hours=60))
```

Исходя из предположения, что вы создадите более одного экземпляра класса и хотите применить дополнение к каждому экземпляру, декораторы обычно будут выигрышным вариантом с точки зрения как размера кода, так и его сопровождения.

Примечание, касающееся нестыковки версий для атрибутов. Предыдущий декоратор отслеживания работает при явном доступе к именам атрибутов во всех версиях Python. Тем не менее, как выяснилось в главах 38, 32 и других местах, метод `__getattr__` перехватывает неявный доступ встроенных операций к методам перегрузки операций наподобие `__str__` и `__repr__` в классических классах Python 2.X, но не в классах нового стиля Python 3.X.

В классах Python 3.X экземпляры наследуют стандартные версии для некоторых, но не всех таких имен из класса (на самом деле из суперкласса `object`). Кроме того, в Python 3.X неявно вызываемые атрибуты для встроенных операций вроде вывода и + не прогоняются через метод `__getattr__` или родственный ему `__getattribute__`. В классах нового стиля встроенные операции начинают поиск в классах, полностью пропуская просмотр нормального экземпляра.

Сказанное означает, что в том виде, как есть, класс оболочки отслеживания, основанный на `__getattr__`, будет автоматически отслеживать и распространять вызовы перегрузки операций для встроенных операций в Python 2.X, но не в Python 3.X. Чтобы увидеть это, отобразим `x` прямо в конце предыдущего интерактивного сеанса — в Python 2.X атрибут `__repr__` отслеживается и список выводится ожидаемым образом, но в Python 3.X отслеживание не происходит и список выводится с использованием стандартного отображения для класса `Wrapper`:

```
>>> x                                   # Python 2.X
Trace: __repr__
[4, 5, 6, 7]
>>> x                                   # Python 3.X
<interfacetracer.Tracer.<locals>.Wrapper object at 0x02946358>
```

Чтобы то же самое работало в Python 3.X, методы перегрузки операций, как правило, должны избыточно переопределяться в классе оболочки вручную, посредством инструментов или за счет определения в суперклассах. Мы снова увидим это в работе при создании декоратора `Private` позже в главе, когда будем исследовать способы добавления методов, требующих такой код, в Python 3.X.

Грубые ошибки, связанные с классами, часть II: предохранение множества экземпляров

Любопытно отметить, что декораторная функция в данном примере может быть *почти* полностью реализована в виде класса с надлежащим протоколом перегрузки операций. Следующая слегка упрощенная версия работает аналогично, т.к. метод `__init__` запускается во время применения к классу декорирования `@`, а метод `__call__` срабатывает при создании экземпляра целевого класса. На этот раз объекты являются экземплярами класса `Tracer`, и мы по существу здесь просто заменяем ссылку из объемлющей области видимости атрибутом экземпляра:

```
class Tracer:
    def __init__(self, aClass):         # При декорировании @
        self.aClass = aClass            # Использование атрибуты экземпляра
    def __call__(self, *args):          # При создании экземпляров
        self.wrapped = self.aClass(*args) # ОДИН (ПОСЛЕДНИЙ) ЭКЗЕМПЛЯР НА КЛАСС!
        return self
```

```
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)
        return getattr(self.wrapped, attrname)

@Tracer                          # Запускается __init__
class Spam:                      # Подобно Spam = Tracer(Spam)
    def display(self):
        print('Spam!' * 8)
...
food = Spam()                    # Запускается __call__
food.display()                   # Запускается __getattr__
```

Однако, как теоретически обсуждалось ранее, такая альтернатива, где используются только классы, поддерживает *множество классов*, но не вполне нормально работает с *множеством экземпляров* заданного класса: каждый вызов, создающий экземпляр, запускает метод `__call__`, который переписывает предыдущий экземпляр. Совокупный эффект заключается в том, что `Tracer` хранит только один экземпляр – последний созданный. Поэкспериментируйте самостоятельно, чтобы понять, в чем дело, но ниже демонстрируется пример проявления проблемы:

```
@Tracer
class Person:                    # Person = Tracer(Person)
    def __init__(self, name):    # Wrapper привязан к Person
        self.name = name
bob = Person('Bob')              # bob - на самом деле экземпляр Wrapper
print(bob.name)                  # Wrapper содержит экземпляр Person
Sue = Person('Sue')
print(sue.name)                  # Экземпляр sue переписывает экземпляр bob
print(bob.name)                  # ПРОБЛЕМА: теперь именем bob оказывается Sue!
```

Вывод кода выглядит следующим образом – поскольку имеется лишь один разделяемый экземпляр декоратора, второй объект переписывает первый:

```
Trace: name
Bob
Trace: name
Sue
Trace: name
Sue
```

Проблема связана с неправильным *хранением состояния* – мы создаем один экземпляр декоратора на класс, но не на экземпляр класса, так что сохраняется только последний экземпляр декоратора. Как было показано в разделе “Грубые ошибки, связанные с классами, часть I: декорирование методов”, решение предусматривает отказ от декораторов на основе классов.

Представленная ранее версия `Tracer`, основанная на функции, *работает* для множества экземпляров, потому что каждый вызов, конструирующий экземпляр, создает новый экземпляр `Wrapper`, а не переписывает состояние единственного разделяемого экземпляра `Tracer`; первоначальная версия без декораторов корректно обрабатывает множество экземпляров по той же самой причине. Мораль здесь в том, что декораторы являются не только магическими, но также могут быть невероятно хитроумными!

Декораторы или управляющие функции

Не считая рассмотренных выше тонкостей, класс декоратора `Tracer` в конечном итоге по-прежнему полагается на метод `__getattr__` при перехвате операций извлечения для внутреннего и внедренного объекта экземпляра. Как выяснилось ранее, все,

что мы фактически достигли — это перенесли вызов, создающий экземпляр, внутрь класса вместо того, чтобы передавать готовый экземпляр управляющей функции. В первоначальном примере отслеживания без декораторов мы бы просто реализовали создание экземпляров по-другому:

```
class Spam:                          # Версия без декоратора
    ...                              # Подойдет любой класс
food = Wrapper(Spam())               # Особый синтаксис создания

@Tracer
class Spam:                          # Версия с декоратором
    ...                              # Для класса требуется синтаксис @
food = Spam()                        # Нормальный синтаксис создания
```

По существу *декораторы классов* перемещают особые синтаксические требования из вызова, создающего экземпляр, в сам оператор `class`. Сказанное также справедливо для рассмотренного ранее примера с классом-одиночкой — вместо декорирования класса и применения нормальных вызовов, создающих экземпляры, мы могли бы просто передавать класс и аргументы для его конструктора управляющей функции:

```
instances = {}
def getInstance(aClass, *args, **kwargs):
    if aClass not in instances:
        instances[aClass] = aClass(*args, **kwargs)
    return instances[aClass]
bob = getInstance(Person, 'Bob', 40, 10)    # В сравнении с
                                            #   bob = Person('Bob', 40, 10)
```

В качестве альтернативы мы могли бы использовать средства интроспекции Python для извлечения сведений о классе из уже созданного экземпляра (при условии, что создание начального экземпляра приемлемо):

```
instances = {}
def getInstance(object):
    aClass = object.__class__
    if aClass not in instances:
        instances[aClass] = object
    return instances[aClass]
bob = getInstance(Person('Bob', 40, 10))    # В сравнении с
                                            #   bob = Person('Bob', 40, 10)
```

То же самое верно для *декораторов функций*: вместо декорирования функции логикой, которая перехватывает последующие вызовы, мы могли бы просто передавать функцию и ее аргументы управляющей функции, координирующей вызов:

```
def func(x, y):                      # Версия без декоратора
    ...                              # def tracer(func, args): ... func(*args)
result = tracer(func, (1, 2))        # Особый синтаксис создания

@tracer
def func(x, y):                      # Версия с декоратором
    ...                              # Повторная привязка имени:
                                     #   func = tracer(func)
result = func(1, 2)                  # Нормальный синтаксис создания
```

Подобный подход с управляющей функцией накладывает бремя использования особого синтаксиса на *вызовы* взамен ожидания синтаксиса декорирования в определениях функций и классов, но также позволяет избирательно применять дополнение на основе вызовов.

Для чего используются декораторы? (Еще раз)

Так почему я только что показал вам, как *не* использовать декораторы для реализации классов-одиночек? В начале главы упоминалось о том, что декораторы преподносят нам компромиссы. Хотя синтаксис имеет значение, все мы слишком часто забываем задавать вопросы "для чего", когда сталкиваемся с новыми инструментами. Теперь, когда вы ознакомились с фактической работой декораторов, давайте отвлечемся на некоторое время, чтобы бегло взглянуть на общую картину, прежде чем снова заняться написанием кода.

Подобно большинству языковых средств декораторам присущи достоинства и недостатки. Скажем, что касается негативных сторон, декораторы могут страдать от трех потенциальных изъянов, которые варьируются в зависимости от типа декоратора.

Изменения типа

Как мы уже видели, когда вставляются оболочки, декорированная функция или класс не предохраняют свой *исходный тип* – происходит повторная привязка к объекту-оболочке (посреднику), что может иметь значение в программах, в которых применяются имена объектов или проверки типов объектов. В примере с классом-одиночкой подходы с декоратором и управляющей функции обеспечивают предохранение исходного типа класса для экземпляров; в примере с отслеживанием ни один подход этого не делает, т.к. требуются оболочки. Конечно, в целом вы должны избегать проверки типов в полиморфных языках наподобие Python, но из большинства правил существуют исключения.

Дополнительные вызовы

Уровень оболочки, добавляемый декорированием, влечет за собой издержки в плане производительности из-за *дополнительного вызова* при каждом обращении к декорированному объекту – вызовы являются относительно затратными по времени операциями, поэтому декорирующие оболочки способны замедлить работу программы. В примере с отслеживанием оба подхода требуют прохождения каждого атрибута через уровень оболочки; в примере с классом-одиночкой дополнительных вызовов удается избежать за счет предохранения исходного типа класса.

Все или ничего

Поскольку декораторы дополняют функцию или класс, они обычно применяются к *каждому* обращению к декорированному объекту в будущем. Это обеспечивает единообразный ввод в действие, но также может оказаться недостатком, если дополнение желательно применять более избирательно на основе вызовов.

Следует отметить, что ни один из перечисленных выше недостатков не считается очень серьезной проблемой. Для большинства программ единообразие декорирования является достоинством, различие между типами вряд ли имеет значение, а влияние на скорость выполнения со стороны дополнительных вызовов оказывается несущественным. Более того, последнее происходит только при использовании оболочек, часто может быть сведено на нет, если мы просто удаляем декоратор, когда требуется оптимальная производительность, и также вытекает из решений без декораторов, где добавляется логика оболочки (включая *метаклассы*, что будет показано в главе 40).

И наоборот, как мы видели в начале главы, декораторы имеют три главных преимущества. Далее описано то, что они предлагают по сравнению с решениями, в которых применяется управляющая (она же "вспомогательная") функция.

Явный синтаксис

Декораторы делают дополнение явным и очевидным. Их синтаксис @ легче опознать, чем специальный код в вызовах, который может появляться в любом месте файла – скажем, в примерах с классом-одиночкой и отслеживанием строки декораторов будут замечены с большей вероятностью, чем добавочный код в вызовах. Кроме того, декораторы позволяют использовать для вызовов, которые создают функции и экземпляры, нормальный синтаксис, знакомый всем программистам на Python.

Сопровождение кода

Декораторы позволяют избежать повторения дополняющего кода при каждом обращении к функции или классу. Поскольку декораторы указываются только один раз в самом определении класса или функции, они устраняют избыточность и упрощают будущее сопровождение кода. В случае класса-одиночки и отслеживания для применения подхода с управляющей функцией нам необходимо использовать специальный код в каждом вызове – дополнительная работа, требуемая как в начале, так и при любых изменениях, которые придется вносить в будущем.

Согласованность

Декораторы снижают вероятность того, что программист забудет применить обязательную логику оболочки. Это порождается главным образом предыдущими двумя преимуществами – поскольку декорирование делается явно и только один раз в самих декорированных объектах, декораторы способствуют более согласованному и единообразному использованию API-интерфейсов, чем специальный код, который должен быть включен в каждый вызов. В примере с классом-одиночкой было бы легко забыть о прогоне всех вызовов, создающих экземпляры класса, через специальный код, что полностью нарушило бы управление одиночкой.

Декораторы также содействуют *инкапсуляции* кода, направленной на сокращение избыточности и минимизацию будущих работ по сопровождению; дополняющий код встречается только раз в вызываемом объекте декоратора, а не копируется для каждого ввода в действие. Хотя добиться этого можно и с помощью управляющих функций, декораторы также предлагают явный синтаксис и бесшовную модель вызовов, что делает их естественным средством для решения задач дополнения.

Тем не менее, получение любого из указанных преимуществ совершенно не требует декораторного синтаксиса, а применение декораторов в итоге является стилистическим выбором. Однако большинство программистов считают их чистым выигрышем, особенно при использовании в качестве инструмента для корректной работы с библиотеками и API-интерфейсами.

Забавная история. Я могу вспомнить похожие доводы за и против функций *конструкторов* в классах – до появления методов `__init__` программисты добивались того же эффекта, пропуская экземпляр через метод вручную при его создании (например, `X=Class().init()`). Тем не менее, несмотря на то, что синтаксис `__init__` по существу являлся лишь стилистическим выбором, со временем он стал универсально предпочитаемым, т.к. был более явным, согласованным и удобным в сопровождении. Естественно, судить исключительно вам, но кажется, что с декораторами связано много аналогичных преимуществ.

Управление функциями и классами напрямую

Большинство примеров в данной главе были спроектированы так, чтобы перехватывать вызовы, создающие функции и экземпляры. Хотя такая роль типична для декораторов, одной только ею они не ограничиваются. Из-за того, что декораторы работают путем прогона новых функций и классов через декораторный код, они также могут применяться для управления самими объектами функций и классов, а не только последующими обращениями к ним.

Предположим, что вы требуете от методов или классов, используемых приложением, регистрации в API-интерфейсе для будущей обработки (может быть, API-интерфейс позже будет обращаться к объектам в ответ на поступление событий). Несмотря на то что вы могли бы предоставить регистрационную функцию, подлежащую вызову вручную после того, как объекты определены, декораторы сделают ваше намерение более явным.

В следующем простом воплощении описанной идеи определяется декоратор, который предназначен для добавления объекта в основанный на словаре реестр и может применяться к функциям *и* классам. Поскольку он возвращает сам объект, а не оболочку, он не перехватывает будущие вызовы:

```
# Регистрация декорированных объектов в API-интерфейсе
from __future__ import print_function      # Python 2.X

registry = {}
def register(obj):                         # Декоратор для классов и функций
    registry[obj.__name__] = obj           # Добавление в реестр
    return obj                             # Возвращение самого объекта, не оболочки

@register
def spam(x):
    return(x ** 2)                         # spam = register(spam)

@register
def ham(x):
    return(x ** 3)

@register
class Eggs:                                # Eggs = register(Eggs)
    def __init__(self, x):
        self.data = x ** 4
    def __str__(self):
        return str(self.data)

print('Registry:')                         # Реестр
for name in registry:
    print(name, '=>', registry[name], type(registry[name]))

print('\nManual calls:')                   # Вызовы вручную
print(spam(2))                             # Обращение к объектам вручную
print(ham(2))                              # Последующие вызовы не перехватываются
X = Eggs(2)
print(X)

print('\nRegistry calls:')                 # Вызовы из реестра
for name in registry:
    print(name, '=>', registry[name](2))      # Обращение из реестра
```

Во время выполнения кода декорированные объекты добавляются в реестр по имени, но при обращении в будущем они по-прежнему работают так, как были первона-

чально реализованы, без прохождения через какой-либо уровень оболочки. На самом деле наши объекты могут запускаться и вручную, и изнутри таблицы реестра:

```
c:\code> py -3 registry-deco.py
Registry:
spam => <function spam at 0x02969158> <class 'function'>
ham => <function ham at 0x02969400> <class 'function'>
Eggs => <class '__main__.Eggs'> <class 'type'>

Manual calls:
4
8
16

Registry calls:
spam => 4
ham => 8
Eggs => 16
```

Такая методика может использоваться, например, в пользовательском интерфейсе при регистрации обработчиков обратных вызовов для действий пользователя. Обработчики могут регистрироваться по имени функции или класса, как было сделано здесь, или же можно было бы применять аргументы декоратора для указания целевого события; за счет добавления оператора `def`, заключающего наш декоратор, можно было бы обеспечить сохранение таких аргументов для использования в декорировании.

Приведенный пример несколько надуман, но применяемая в нем методика чрезвычайно универсальна. Скажем, декораторы функций также можно использовать для обработки атрибутов функции, а декораторы классов могут динамически вставлять новые атрибуты класса и даже новые методы. Рассмотрим показанные ниже декораторы функций – они присваивают атрибутам функции значения с целью последующего применения API-интерфейсом, но не добавляют уровень оболочки для перехвата будущих вызовов:

```
# Дополнение декорированных объектов напрямую

>>> def decorate(func):
        func.marked = True          # Присваивание атрибуту функции значения
                                    #   для будущего использования
        return func

>>> @decorate
    def spam(a, b):
        return a + b

>>> spam.marked
True

>>> def annotate(text):   # То же самое, но значением является аргумент декоратора
        def decorate(func):
            func.label = text
            return func
        return decorate

>>> @annotate('spam data')
    def spam(a, b):                 # spam = annotate(...)(spam)
        return a + b

>>> spam(1, 2), spam.label
(3, 'spam data')
```

Такие декораторы дополняют функции и классы напрямую, не перехватывая обращения к ним в будущем. В следующей главе будет предложено больше примеров декораторов классов, управляющих классами напрямую, т.к. это затрагивает предметную область *метаклассов*; в остатке текущей главы мы проработаем два более крупных учебных примера использования декораторов.

Пример: "закрытые" и "открытые" атрибуты

В последних двух разделах главы представлены более крупные примеры применения декораторов. Оба они сопровождаются минимальным описанием, отчасти из-за того, чтобы уместить главу в разрешенные рамки, но главным образом потому, вы уже должны достаточно хорошо понимать основы декораторов, чтобы самостоятельно разобраться в коде примеров. Будучи универсальными инструментами, предложенные примеры дадут вам шанс увидеть, как концепции декораторов объединяются в более полезном коде.

Реализация закрытых атрибутов

Разрабатываемый пример *декоратора классов* реализует объявление `Private` для атрибутов класса, т.е. атрибутов, хранящихся в экземпляре или унаследованных из его суперклассов. Объявление `Private` запрещает доступ к таким атрибутам с целью извлечения и изменения *извне* декорированного класса, но разрешает самому классу свободно обращаться к этим именам внутри собственных методов. Здесь не воспроизводятся в точности средства языка C++ или Java, но обеспечивается похожий контроль доступа как вариант в Python.

В главе 30 мы встречали незавершенную пробную реализацию защиты атрибутов экземпляра только от *изменений*. В разрабатываемой здесь версии концепция расширяется с целью проверки допустимости также операций *извлечения* и для реализации модели вместо наследования используется делегирование. В определенном смысле фактически это всего лишь расширение декоратора классов для отслеживания атрибутов, который мы видели ранее.

Хотя в примере для реализации защиты атрибутов применяется новый синтаксический сахар декораторов классов, перехват атрибутов в конечном итоге основан на методах перегрузки операций `__getattr__` и `__setattr__`, с которыми мы сталкивались в предшествующих главах. Когда обнаруживается операция доступа к закрытому атрибуту, в данной версии используется оператор `raise` для генерации исключения наряду с выдачей сообщения об ошибке; исключение можно перехватить в операторе `try` или разрешить сценарию закончить свою работу.

Ниже представлен код декоратора вместе с кодом самотестирования в конце файла. Он будет работать в Python 3.X и 2.X (в Python 2.6 и последующих версиях), т.к. задействует нейтральный к версиям синтаксис `print` и `raise`. Правда, в таком виде, как есть, он перехватывает отправку встроенных операций атрибутам методов перегрузки операций только в Python 2.X (вскоре мы обсудим это более подробно):

```
"""
Файл access1.py (Python 3.X + 2.X)

Защита для атрибутов, извлекаемых из экземпляров класса.
Пример использования приведен в коде самотестирования в конце файла.

Декоратор такой же, как Doubler = Private('data', 'size')(Doubler).
Private возвращает onDecorator, onDecorator возвращает onInstance,
а в каждом экземпляре onInstance внедрен экземпляр Doubler.
"""
```

```
traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def Private(*privates):              # privates в объемлющей области видимости
    def onDecorator(aClass):         # aClass в объемлющей области видимости
        class onInstance:            # wrapped в атрибуте экземпляра
            def __init__(self, *args, **kargs):
                self.wrapped = aClass(*args, **kargs)

            def __getattr__(self, attr):      # Для собственных атрибутов
                                              #   getattr не вызывается
                trace('get:', attr)  # Предполагается, что остальные внутри wrapped
                if attr in privates:
                    raise TypeError('private attribute fetch: ' + attr)
                else:
                    return getattr(self.wrapped, attr)

            def __setattr__(self, attr, value):    # Операции доступа извне
                trace('set:', attr, value)    # Остальные выполняются нормально
                if attr == 'wrapped':         # Разрешить доступ к собственным
                                              #   атрибутам
                    self.__dict__[attr] = value  # Избежать зацикливания
                elif attr in privates:
                    raise TypeError('private attribute change: ' + attr)
                else:
                    setattr(self.wrapped, attr, value) # Атрибуты внутреннего объекта
        return onInstance                     # Либо использовать __dict__
    return onDecorator

if __name__ == '__main__':
    traceMe = True

    @Private('data', 'size')              # Doubler = Private(...)(Doubler)
    class Doubler:
        def __init__(self, label, start):
            self.label = label        # Операции доступа внутри целевого класса
            self.data = start         # Не перехватываются: выполняются нормально
        def size(self):
            return len(self.data)     # Методы запускаются без какой-либо проверки
        def double(self):             # Потому что защита не наследуется
            for i in range(self.size()):
                self.data[i] = self.data[i] * 2
        def display(self):
            print('%s => %s' % (self.label, self.data))

X = Doubler('X is', [1, 2, 3])
Y = Doubler('Y is', [-10, -20, -30])

# Весь следующий код выполняется успешно
print(X.label)                       # Операции доступа извне целевого класса
X.display(); X.double(); X.display()    # Перехватываются: проверяются,
                                        #   делегируются
print(Y.label)
Y.display(); Y.double()
Y.label = 'Spam'
Y.display()

# Весь следующий код должным образом терпит неудачу
"""
```

```
print(X.size())       # Выводится TypeError: private attribute fetch: size
print(X.data)
X.data = [1, 1, 1]
X.size = lambda S: 0
print(Y.data)
print(Y.size())
"""
```

Когда traceMe равно True, код самотестирования файла модуля производит приведенный далее вывод. Обратите внимание, что декоратор перехватывает и проверяет допустимость операций извлечения и присваивания атрибутов, выполненных *вне* внутреннего класса, но не перехватывает операции доступа к атрибутам *внутри* самого класса:

```
c:\code> py -3 access1.py
[set: wrapped <__main__.Doubler object at 0x00000000029769B0>]
[set: wrapped <__main__.Doubler object at 0x00000000029769E8>]
[get: label]
X is
[get: display]
X is => [1, 2, 3]
[get: double]
[get: display]
X is => [2, 4, 6]
[get: label]
Y is
[get: display]
Y is => [-10, -20, -30]
[get: double]
[set: label Spam]
[get: display]
Spam => [-20, -40, -60]
```

Детали реализации, часть I

Код немного сложен, и вероятно вам лучше изучить его самостоятельно, чтобы увидеть, как он работает. Далее приведено несколько примечаний, которые призваны помочь.

Наследование или делегирование

В первоначальном примере защиты из главы 30 с применением *наследования* метод `__setattr__` подмешивался для перехвата операций доступа. Однако наследование затрудняет это, т.к. задача различения операций доступа изнутри и снаружи класса не является простой (доступ изнутри должен быть разрешен, а доступ снаружи ограничен). Чтобы обойти проблему, пример в главе 30 требовал использования наследуемыми классами операций присваивания словаря `__dict__` для установки атрибутов — в лучшем случае незаконченное решение.

В текущей версии вместо наследования применяется *делегирование* (внедрение одного объекта внутрь другого); такая схема лучше подходит для нашей задачи, поскольку она значительно облегчает различение операций доступа изнутри и снаружи целевого класса. Операции доступа снаружи целевого класса перехватываются методами перегрузки операций из уровня оболочки; в случае допустимости их выполнение делегируется классу. Операции доступа внутри самого класса (т.е. инициируемые через `self` в коде методов класса) не перехватываются и выполняются нормально без проверки, потому что защита в данной версии не наследуется.

Аргументы декоратора

Реализованный здесь декоратор класса принимает любое количество аргументов для указания имен закрытых атрибутов. Тем не менее, в действительности эти аргументы передаются функции `Private`, которая возвращает декораторную функцию, предназначенную для применения к целевому классу. То есть аргументы используются даже до того, как происходит декорирование; `Private` возвращает декоратор, который в свою очередь "запоминает" список закрытых атрибутов как ссылку из объемлющей области видимости.

Предохранение состояния и объемлющие области видимости

Что касается объемлющих областей видимости, то в коде на самом деле есть *три уровня* предохранения состояния.

- Аргументы для `Private` задействуются до того, как происходит декорирование, и сохраняются в виде ссылок из объемлющей области видимости с целью применения в `onDecorator` и `onInstance`.
- Аргумент класса для `onDecorator` используется во время декорирования и сохраняется как ссылка из объемлющей области видимости с целью применения при создании экземпляра.
- Внутренний объект экземпляра сохраняется как атрибут экземпляра в посреднике `onInstance` для использования при последующем доступе к атрибутам снаружи класса.

Все это работает вполне естественно с учетом правил Python, касающихся областей видимости и пространств имен.

Использование `__dict__` и `__slots__` (и других виртуальных имен)

При установке собственного атрибута `wrapped` в `onInstance` метод `__setattr__` опирается на словарь пространств имен атрибутов `__dict__` объекта экземпляра. Как выяснилось в предыдущей главе, данный метод не может присваивать значение атрибуту напрямую, не приводя к зацикливанию. Однако для установки атрибутов в самом *внутреннем* объекте он применяет встроенную функцию `setattr` вместо `__dict__`. Кроме того, для извлечения атрибутов из внутреннего объекта используется встроенная функция `getattr`, т.к. они могут храниться в самом объекте или наследоваться им.

По этой причине код будет работать для большинства классов, в том числе тех, которые содержат "виртуальные" атрибуты, основанные на *слотах*, *свойствах*, *дескрипторах* и даже `__getattr__` с прочими вариантами. Допуская применение словаря пространств имен только для себя, а также используя нейтральные к хранилищу инструменты для внутреннего объекта, класс оболочки избегает ограничений, присущих другим инструментам.

Например, как обсуждалось в главе 32, классы нового стиля, содержащие `__slots__`, могут не хранить атрибуты в словаре `__dict__` (и на самом деле могут даже не иметь его вообще). Тем не менее, поскольку здесь мы полагаемся на `__dict__` только на уровне `onInstance`, но не во внутреннем экземпляре, такая проблема не возникает. Вдобавок из-за того, что встроенные функции `setattr` и `getattr` применяются к атрибутам, основанным на `__dict__` и `__slots__`, наш декоратор применим к классам, использующим любую из двух схем хранения. По той же причине декоратор применим к свойствам нового стиля и подобным инструментам: делегированные имена снова будут искаться во внутреннем экземпляре независимо от атрибутов самого объекта декоратора.

Обобщение также для открытых объявлений

Располагая реализацией `Private`, код несложно обобщить, чтобы сделать возможными также открытые объявления (`Public`) — они по существу являются противоположностью объявлений `Private`, и потому понадобится лишь инвертировать внутреннюю проверку. Представленный в этом разделе пример позволяет классу использовать декораторы для определения набора атрибутов экземпляров `Private` или `Public`, т.е. атрибутов любого вида, хранящихся в экземпляре либо унаследованных из его классов, с описанной ниже семантикой.

- Декоратор `Private` объявляет атрибуты экземпляров класса, которые нельзя извлекать или присваивать им значения кроме как внутри кода методов класса. Таким образом, любое имя, объявленное как `Private`, не будет доступным извне класса, но любое имя, не объявленное как `Private`, может свободно извлекаться и присваиваться за пределами класса.
- Декоратор `Public` объявляет атрибуты экземпляров класса, которые можно извлекать и присваивать им значения как снаружи класса, так и внутри методов класса. То есть любое имя, объявленное как `Public`, доступно везде, но обращаться извне класса к любому имени, не объявленному как `Public`, нельзя.

Объявления `Private` и `Public` должны быть взаимоисключающими: когда применяется `Private`, то все необъявленные имена считаются `Public`, а когда используется `Public`, то все необъявленные имена считаются `Private`. В сущности, они являются противоположностями, хотя необъявленные имена, не созданные методами класса, ведут себя несколько по-другому — новым именам можно присваивать значения и соответственно создавать их за пределами класса в случае объявления `Private` (все необъявленные имена доступны), но не `Public` (все необъявленные имена недоступны).

Самостоятельно изучите приведенный далее код, чтобы получить представление о том, как он работает. Обратите внимание, что эта схема добавляет дополнительный *четвертый уровень предохранения состояния* поверх описанных в предыдущем разделе: проверочные функции, применяемые определениями `lambda`, сохраняются в дополнительной объемлющей области видимости. Пример реализован для выполнения под управлением Python 3.X или 2.X (Python 2.6 или последующих версий), хотя с ним связано одно предостережение при запуске в Python 3.X (которое кратко объясняется в строке документации самого файла и более развернуто после листинга кода):

```
"""
Файл access2.py (Python 3.X + 2.X)
Декораторы классов с объявлениями атрибутов Private и Public.

Управляют внешним доступом к атрибутам, хранящимся в экземпляре или
унаследованным из его классов.
Private объявляет имена атрибутов, которые нельзя извлекать или присваивать
им значения за пределами декорируемого класса, а Public объявляет все имена,
к которым можно применять упомянутые действия.

Предостережение: в Python 3.X это работает только для явно именованных
атрибутов: в классах
нового стиля методы перегрузки операций __X__, неявно запускаемые для
встроенных операций,
не вызывают ни __getattr__, ни __getattribute__. Чтобы перехватывать и
делегировать выполнение
встроенных операций, необходимо добавить здесь методы __X__.
"""
```

```
traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)

            def __getattr__(self, attr):
                trace('get:', attr)
                if failIf(attr):
                    raise TypeError('private attribute fetch: ' + attr)
                                    # извлечение закрытого атрибута
                else:
                    return getattr(self.__wrapped, attr)

            def __setattr__(self, attr, value):
                trace('set:', attr, value)
                if attr == '_onInstance__wrapped':
                    self.__dict__[attr] = value
                elif failIf(attr):
                    raise TypeError('private attribute change: ' + attr)
                                    # изменение закрытого атрибута
                else:
                    setattr(self.__wrapped, attr, value)
        return onInstance
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))
```

Один из сценариев использования можно найти в коде самотестирования предыдущего примера. Ниже демонстрируется применение декораторов классов в интерактивной подсказке; он работают одинаково в Python 2.X и 3.X для атрибутов, ссылка на которые производится по явным именам. Как было заявлено, имена не Private либо имена Public можно извлекать и изменять их значения извне целевого класса, но имена Private либо имена не Public — нельзя:

```
>>> from access2 import Private, Public

>>> @Private('age')                       # Person = Private('age')(Person)
    class Person:                         # Person = onInstance с состоянием
        def __init__(self, name, age):
            self.name = name
            self.age = age          # Операции доступа изнутри работают нормально

>>> X = Person('Bob', 40)
>>> X.name                          # Операции доступа снаружи проверяются
'Bob'
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch: age
Ошибка типа: извлечение закрытого атрибута: age
```

```
>>> X.age = 'Tom'
TypeError: private attribute change: age
Ошибка типа: изменение закрытого атрибута: age

>>> @Public('name')
    class Person:
        def __init__(self, name, age):
            self.name = name
            self.age = age

>>> X = Person('bob', 40)                       # X - onInstance
>>> X.name                                      # onInstance включает Person
'bob'
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch: age
Ошибка типа: извлечение закрытого атрибута: age
>>> X.age = 'Tom'
TypeError: private attribute change: age
Ошибка типа: изменение закрытого атрибута: age
```

Детали реализации, часть II

Далее представлено несколько финальных замечаний по данной версии, которые помогут проанализировать код. Поскольку это всего лишь обобщение версии из предыдущего раздела, здесь также применимы ранее приведенные соображения относительно реализации.

Использование псевдозакрытых имен __X

Помимо обобщения в текущей версии также задействовано средство корректировки псевдозакрытых имен __X (с которым мы сталкивались в главе 31), чтобы сделать атрибут wrapped локальным для управляющего класса-посредника за счет снабжения его префиксом в виде имени упомянутого класса. В итоге удается избежать присущего предыдущей версии риска конфликта имен с атрибутом wrapped, который может присутствовать в реальном внутреннем классе, что полезно для универсального инструмента подобного рода. Однако мы получаем не совсем полноценную "защиту", т.к. скорректированная версия имени может свободно применяться за пределами класса. Имейте в виду, что мы также обязаны использовать полностью расширенную строку имени — '_onInstance__wrapped' — в качестве проверочного значения в __setattr__, поскольку именно таким образом Python ее изменяет.

Нарушение защиты

Несмотря на то что в примере реализовано средство управления доступом для атрибутов экземпляра и его классов, его можно обойти различными способами — скажем, явно указывая расширенную версию атрибута wrapped (bob.pay может не сработать, но полностью скорректированное имя bob._onInstance__wrapped.pay — вполне!). Тем не менее, если вам приходится поступать так явно, то средства управления доступом, вероятно, окажется достаточно для нормального предполагаемого применения. Конечно, в целом средство управления защитой можно обходить и в других языках, приложив должные усилия (#define private public может работать в ряде реализаций C++). Хотя средство управления доступом способно уменьшить случайные

изменения, по большей части это зависит от программистов на любом языке; всякий раз, когда можно изменять исходный код, управление доступом, не имеющее слабых мест, всегда будет лишь несбыточной мечтой.

Компромиссы, связанные с декораторами

Мы снова могли бы получить те же самые результаты без декораторов, используя управляющие функции или реализуя повторную привязку имен декораторов вручную; однако, декораторный синтаксис делает решение согласованным и чуть более очевидным в коде. Основные недостатки такого и других подходов, основанных на оболочках, заключаются в том, что доступ к атрибутам сопровождается дополнительным вызовом, а экземпляры декорированных классов в действительности не являются экземплярами исходных классов, которые были декорированы. Например, если вы проверите их тип с помощью `X.__class__` или `isinstance(X, C)`, то обнаружите, что они оказываются экземплярами *класса-оболочки*. Тем не менее, если только вы не планируете выполнять интроспекцию типов объектов, тогда указанная проблема с типом вряд ли будет существенной, а дополнительный вызов может быть применим главным образом к стадии разработки; как выяснится позже, доступны способы автоматического удаления декорирования, когда оно нежелательно.

Нерешенные проблемы

В имеющемся виде код примера работает в Python 2.X и 3.X для методов, вызываемых явно по имени, как и планировалось. Однако подобно большинству программного обеспечения всегда есть возможности для усовершенствования. Наиболее заметно то, что данный инструмент приводит к неоднозначным показателям производительности для методов перегрузки операций, если они используются клиентскими классами.

В том виде, как есть, класс-посредник представляет собой классический класс при запуске под управлением Python 2.X, но класс нового стиля, когда выполняется в Python 3.X. Будучи таковым, код поддерживает любой клиентский класс в Python 2.X, но в Python 3.X терпит неудачу с проверкой доступа или делегированием работы методам перегрузки операций, что неявно инициируется встроенными операциями, если только они не переопределены в посреднике. Клиенты, которые не задействуют перегрузку операций, поддерживаются полноценно, но остальные могут требовать написания дополнительного кода в Python 3.X.

Важно отметить, что здесь речь идет не о проблеме классов нового стиля, а о проблеме, связанной с *версиями* Python — один и тот же код выполняется по-другому и терпит неудачу только в Python 3.X. Так как природа класса внутреннего объекта для посредника несущественна, нас интересует только собственный код посредника, который работает в Python 2.X и не работает в Python 3.X.

Мы уже несколько раз сталкивались с этой проблемой в книге, но давайте кратко рассмотрим ее влияние на написанный ранее очень реалистичный код и изучим обходной путь.

Предостережение: неявный запуск методов перегрузки операций не работает в Python 3.X

Как и все основанные на делегировании классы, в которых применяется метод `__getattr__`, наш декоратор работает независимо от версии только для нормально именованных или явно вызываемых атрибутов. При неявном запуске встроенными операциями методы перегрузки операций вроде `__str__` и `__add__` для классов но-

вого стиля работают иначе. Из-за того, что в Python 3.X код интерпретируется как класс нового стиля, в текущей реализации, выполняемой под управлением Python 3.X, таким операциям не удается добраться до внутреннего объекта, где они определены.

В предыдущей главе было показано, что встроенные операции ищут имена методов перегрузки операций внутри *экземпляров* в случае классических классов, но не классов нового стиля. Для последних они полностью пропускают экземпляр и начинают поиск нужных методов в *классах* (формально внутри словарей пространств имен всех классов в дереве наследования). Следовательно, методы перегрузки операций `__X__`, неявно запускаемые для встроенных операций, в классах нового стиля *не* инициируют выполнение ни `__getattr__`, ни `__getattribute__`; поскольку при операциях извлечения атрибутов подобного рода метод `__getattr__` нашего класса `onInstance` вообще пропускается, они не могут быть проверены или делегированы.

Наш класс декоратора не реализован явно как класс нового стиля (за счет его наследования от `object`), поэтому он будет перехватывать обращение к методам перегрузки операций, если запущен под управлением Python 2.X как стандартный классический класс. Тем не менее, из-за того, что в Python 3.X все классы автоматически становятся классами нового стиля (что обязательно), такие методы *не будут* работать, если они реализованы внутренним объектом — посредник их не перехватывает, а потому и не передает.

Самый непосредственный обходной способ в Python 3.X предусматривает избыточное переопределение в классе `onInstance` всех методов перегрузки операций, которые вероятно будут использоваться во внутренних объектах. Такие дополнительные методы можно добавлять вручную, с помощью инструментов, частично автоматизирующих задачу (скажем, посредством декораторов классов или обсуждаемых в следующей главе метаклассов), либо путем их определения в многократно применяемых суперклассах. Несмотря на утомительность и насыщенность в плане кода, достаточную для того, чтобы по большому счету не рассматривать здесь подходы, удовлетворяющие данное требование в Python 3.X, вскоре мы все же исследуем их.

Однако сначала имеет смысл осознать отличие, для чего попробовать применить декоратор к классу, который использует методы перегрузки операций, в Python 2.X. Проверка допустимости работает, как прежде, а метод `__str__`, запускаемый при выводе, и метод `__add__`, вызываемый для операции `+`, обращаются к методу `__getattr__` декоратора, в итоге приводя к проверке и корректному делегированию работы целевому объекту `Person`:

```
C:\code> c:\python27\python
>>> from access2 import Private
>>> @Private('age')
    class Person:
        def __init__(self):
            self.age = 42
        def __str__(self):
            return 'Person: ' + str(self.age)
        def __add__(self, yrs):
            self.age += yrs

>>> X = Person()
>>> X.age                      # Проверка имени корректно не проходит
TypeError: private attribute fetch: age
Ошибка типа: извлечение закрытого атрибута: age
>>> print(X)                   # __getattr__ => запускается Person.__str__
Person: 42
```

```
>>> X + 10                        # __getattr__ => запускается Person.__add__
>>> print(X)                      # __getattr__ => запускается Person.__str__
Person: 52
```

Тем не менее, когда тот же код запускается под управлением Python 3.X, неявно вызываемые методы `__str__` и `__add__` пропускают метод `__getattr__` декоратора и ищут определения в самом классе декоратора или выше него; `print` находит стандартный метод отображения, унаследованный из типа класса (формально из подразумеваемого суперкласса `object` в Python 3.X), а + генерирует ошибку из-за того, что никакой стандартный метод не был унаследован:

```
C:\code> c:\python37\python
>>> from access2 import Private
>>> @Private('age')
    class Person:
       def __init__(self):
          self.age = 42
       def __str__(self):
          return 'Person: ' + str(self.age)
       def __add__(self, yrs):
          self.age += yrs
>>> X = Person()                 # Проверка имени по-прежнему работает
>>> X.age    # Но в Python 3.X не удается делегирование встроенных операций!
TypeError: private attribute fetch: age
Ошибка типа: извлечение закрытого атрибута: age
>>> print(X)
<access2.accessControl.<locals>.onDecorator.<locals>.onInstance object at
...etc>
>>> X + 10
TypeError: unsupported operand type(s) for +: 'onInstance' and 'int'
Ошибка типа: неподдерживаемые типы операндов для +: onInstance и int
>>> print(X)
<access2.accessControl.<locals>.onDecorator.<locals>.onInstance object at
...etc>
```

Как ни странно, подобное происходит только при координировании выполнения встроенных операций; явные вызовы методов перегрузки направляются `__getattr__`, хотя от клиентов, применяющих перегрузку операций, нельзя ожидать того же:

```
>>> X.__add__(10)                 # Хотя вызовы по имени работают нормально
>>> X._onInstance__wrapped.age   # Нарушение защиты для просмотра результата...
52
```

Другими словами, это вопрос *встроенных операций в сравнении с явными вызовами*; он имеет мало общего с действительными именами затрагиваемых методов. Просто для встроенных операций интерпретатор Python пропускает шаг в случае классов нового стиля Python 3.X.

Использование альтернативного метода `__getattribute__` здесь не поможет — несмотря на то, что он определяется для перехвата ссылки на каждый атрибут (а не только неопределенных имен), он не запускается встроенными операциями. Средство свойств Python, которое обсуждалось в главе 38, тоже напрямую не поможет; вспомните, что свойства автоматически запускают код, ассоциированный с *конкретными* атрибутами, которые определялись при реализации класса, и не предназначены для обработки произвольных атрибутов во внутренних объектах.

Подходы к переопределению методов перегрузки операций для Python 3.X

Как упоминалось ранее, наиболее прямолинейным решением в Python 3.X является избыточное переопределение имен методов перегрузки операций, которые могут появляться во внутренних объектах в основанных на делегировании классах, подобных нашему декоратору. Прием не идеален, потому что он создает некоторую избыточность кода особенно по сравнению с решениями для Python 2.X. Однако его нельзя считать нереально объемной попыткой написания кода; он может быть до определенной степени автоматизирован с помощью инструментов или суперклассов, достаточен для того, чтобы заставить наш декоратор работать в Python 3.X, и способен разрешить объявлять имена методов перегрузки операций как `Private` или `Public`, предполагая наличие запуска внутри методов перегрузки проверок `failIf`.

Встраиваемое определение. Например, ниже демонстрируется подход со *встраиваемым* переопределением – добавление к классу-посреднику переопределения для каждого метода перегрузки операций, который может определять внутренний объект, чтобы перехватывать их и делегировать. В целях иллюстрации мы добавили всего лишь четыре метода перехвата операций, но остальные аналогичны (новый код выделен полужирным):

```
def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)
            # Перехватывать и делегировать встроенные операции особым образом
            def __str__(self):
                return str(self.__wrapped)
            def __add__(self, other):
                return self.__wrapped + other      # Или getattr(x, '__add__')(y)
            def __getitem__(self, index):
                return self.__wrapped[index]       # При необходимости
            def __call__(self, *args, **kargs):
                return self.__wrapped(*args, **kargs)    # При необходимости
            # плюс любые другие, которые нужны

            # Перехватывать и делегировать доступ к атрибутам
            #   по имени обобщенным образом
            def __getattr__(self, attr): ...
            def __setattr__(self, attr, value): ...
        return onInstance
    return onDecorator
```

Подмешиваемые суперклассы. В качестве альтернативы методы можно вставлять посредством общего *суперкласса* – с учетом того, что методов подобного рода может быть много (десятки), внешний класс часто лучше подходит для решения задачи, особенно если он достаточно универсален, чтобы применяться в любом таком интерфейсном классе-посреднике. Для перехвата и делегирования встроенных операций достаточно будет одной из следующих двух схем с подмешиваемыми классами (вероятно среди прочих).

- Первая схема перехватывает встроенные операции и принудительно перенаправляет их методу `__getattr__` подкласса. Она требует, чтобы имена методов перегрузки операций были открытыми для спецификаций каждого декоратора, но обращения к встроенным операциям будут работать аналогично явным вызовам по именам и классическим классам Python 2.X.

- Вторая схема перехватывает встроенные операции и перенаправляет их прямо внутреннему объекту. Она требует наличия доступа и предполагает, что атрибут класса-посредника по имени `_wrapped` предоставит доступ к внедренному объекту. Подход далек от идеала, т.к. он препятствует использованию внутренними объектами того же имени и создает зависимость от подкласса, но лучше применения скорректированного и специфичного для класса атрибута `_onInstance__wrapped` и не хуже аналогично именованного метода.

Как и подход со встраиваемым определением, обе схемы с подмешиваемыми классами также требуют по одному методу на встроенную операцию в универсальных инструментах, которые являются посредниками для интерфейсов произвольных объектов. Обратите внимание, что эти классы перехватывают *обращение* к операциям, а не *извлечение* атрибутов операций, и потому должны выполнять действительные операции, делегируя обработку вызова или выражения:

```
class BuiltinsMixin:
    def __add__(self, other):
        return self.__class__.__getattr__(self, '__add__')(other)
    def __str__(self):
        return self.__class__.__getattr__(self, '__str__')()
    def __getitem__(self, index):
        return self.__class__.__getattr__(self, '__getitem__')(index)
    def __call__(self, *args, **kargs):
        return self.__class__.__getattr__(self, '__call__')(*args, **kargs)
    # плюс любые другие, которые нужны

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            ...остальной код не изменился...
            def __getattr__(self, attr): ...
            def __setattr__(self, attr, value): ...

class BuiltinsMixin:
    def __add__(self, other):
        return self._wrapped + other          # Предполагает наличие _wrapped
    def __str__(self):                        # Пропускает __getattr__
        return str(self._wrapped)
    def __getitem__(self, index):
        return self._wrapped[index]
    def __call__(self, *args, **kargs):
        return self._wrapped(*args, **kargs)
    # плюс любые другие, которые нужны

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            ...и использовать self._wrapped вместо self.__wrapped...
            def __getattr__(self, attr): ...
            def __setattr__(self, attr, value): ...
```

И тот, и другой подмешиваемый суперкласс будет посторонним кодом, но должен быть реализован только один раз и выглядит гораздо более прямолинейным, чем разнообразные подходы с инструментами на базе *метаклассов* или *декораторов*, которые избыточно заполняют класс-посредник необходимыми методами (принципы, лежащие в основе таких инструментов, демонстрируются в примерах дополнения классов в главе 40).

Вариации кода: перенаправляющие классы, дескрипторы, автоматизация. Естественно, оба подмешиваемых суперкласса из предыдущего раздела могут быть улучшены за счет внесения в код дополнительных изменений, которые в основном мы здесь опустим за исключением двух вариаций, заслуживающих краткого упоминания. Во-первых, взгляните на следующую модификацию первого подмешиваемого класса, где используется более простая кодовая структура, но которая влечет за собой добавочный вызов для каждой встроенной операции, снижая скорость ее выполнения (хотя возможно и не настолько значительно в контексте посредника):

```
class BuiltinsMixin:
    def reroute(self, attr, *args, **kargs):
        return self.__class__.__getattr__(self, attr)(*args, **kargs)

    def __add__(self, other):
        return self.reroute('__add__', other)
    def __str__(self):
        return self.reroute('__str__')
    def __getitem__(self, index):
        return self.reroute('__getitem__', index)
    def __call__(self, *args, **kargs):
        return self.reroute('__call__', *args, **kargs)
    # плюс любые другие, которые нужны
```

Во-вторых, все предшествующие подмешиваемые классы для встроенных операций реализуют каждый метод перегрузки операции *явным образом* и перехватывают *вызов*, выданный для операции. В альтернативной версии мы могли бы взамен автоматически *генерировать* методы из списка имен и перехватывать только *извлечение* атрибута, предваряющее вызов, за счет создания *дескрипторов* уровня класса, которые рассматривались в предыдущей главе. Например, как и во второй альтернативной версии подмешиваемого класса, в приведенной ниже вариации предполагается, что внутренний объект имеет имя `_wrapped` в самом экземпляре класса-посредника:

```
class BuiltinsMixin:
    class ProxyDesc(object):                    # object для Python 2.X
        def __init__(self, attrname):
            self.attrname = attrname
        def __get__(self, instance, owner):
            return getattr(instance._wrapped, self.attrname)  # Предполагается
                                                              #   _wrapped

    builtins = ['add', 'str', 'getitem', 'call']    # Плюс любые другие
    for attr in builtins:
        exec('__%s__ = ProxyDesc("__%s__")' % (attr, attr))
```

Такая вариация кода может оказаться самой компактной, но также наиболее неявной и сложной, а вдобавок она тесно связана со своими подклассами по разделяемому имени. Цикл в конце класса эквивалентен показанному далее коду, выполняемому в локальной области видимости подмешиваемого класса — он создает дескрипторы, которые реагируют на первоначальный поиск имен их извлечением из внутреннего объекта в методе `__get__` вместо того, чтобы перехватывать поступающий позже вызов операции:

```
__add__ = ProxyDesc("__add__")
__str__ = ProxyDesc("__str__")
...и так далее...
```

Благодаря добавлению таких методов перегрузки операций (либо путем встраивания, либо за счет наследования от подмешиваемого класса) предыдущий пример клиента `Private`, где операции + и `print` перегружались с помощью `__str__` и `__add__`, корректно работает под управлением Python 2.X и 3.X, как и подклассы, которые перегружают индексирование и вызовы. При желании провести дальнейшие эксперименты ищите полный исходный код всех вариантов в файлах `access2_builtins*.py` внутри пакета примеров для книги; в ответе на один из контрольных вопросов в конце главы будет задействована еще и третья вариация подмешиваемого класса `BuiltinsMixin`.

Должны ли проверяться методы перегрузки операций?

Добавление поддержки для методов перегрузки операций в общем случае требует интерфейсных классов-посредников, чтобы правильно делегировать вызовы. Тем не менее, в нашем специфическом приложении защиты также возникает ряд дополнительных проектных вариантов. В частности, защита доступа к методам перегрузки операций отличается в зависимости от реализации.

- Из-за вызова `__getattr__` перенаправляющие подмешиваемые классы требуют, чтобы либо все доступные имена `__X__` были указаны в объявлениях `Public`, либо взамен применять `Private`, когда в клиентах присутствует перегрузка операций. В классах, интенсивно использующих перегрузку, дескриптор `Public` может оказаться непрактичным.
- Из-за полного игнорирования `__getattr__`, как здесь реализовано, подмешиваемые классы со встраиванием и `self._wrapped` не обладают такими ограничениями, но они не позволяют делать встроенные операции закрытыми. Вдобавок они приводят к тому, что координирование встроенных операций работает неодинаково при явных вызовах методов `__X__` по имени и в стандартных классических классах Python 2.X.
- Классическим классам Python 2.X присущи ограничения, описанные в первом пункте списка, просто потому, что все имена `__X__` автоматически проходят через метод `__getattr__`.
- Имена методов и протоколы перегрузки операций отличаются между линейками Python 2.X и Python 3.X, что затрудняет декорирование, по-настоящему не зависящее от версии (скажем, декораторы `Public` могут нуждаться в перечислении имен из обеих линеек).

Мы не будем здесь формулировать окончательную политику, но отметим, что некоторые интерфейсные классы-посредники могут разрешать именам методов операций `__X__` во время делегирования всегда проходить без проверки.

Однако в общем случае приспособление классов нового стиля Python 3.X в качестве посредников при делегировании требует значительного объема дополнительного кода. В принципе для универсального инструмента, к каковым относится декоратор защиты, *каждый* метод перегрузки операции, который больше автоматически не координируется как нормальный атрибут экземпляра, будет нуждаться в избыточном определении. Вот почему мы опустили это расширение в своем коде: существует более 50 таких методов! Поскольку все классы в Python 3.X являются классами нового стиля, писать основанный на делегировании код становится сложнее, хотя и необязательно невозможно.

Альтернативные реализации: вставка метода `__getattribute__`, инспектирование стека вызовов

Несмотря на то что избыточное определение методов перегрузки операций в классах оболочек, вероятно, считается самым простым путем выхода из затруднительного положения в Python 3.X, обрисованного в предыдущем разделе, он вовсе не единственный. Нам не хватит здесь места для дальнейшего исследования данной проблемы, поэтому вы должны самостоятельно провести более глубокий анализ. Тем не менее, поскольку одна тупиковая альтернатива хорошо иллюстрирует концепции, лежащие в основе классов, она заслуживает краткого упоминания.

Один недостаток, присущий примеру с защитой, заключается в том, что объекты экземпляров являются не подлинными экземплярами исходного класса, а экземплярами класса оболочки. Для поддержки таких случаев мы могли бы попробовать достичь похожего эффекта, *вставив* в исходный класс методы `__getattribute__` и `__setattr__`, чтобы перехватывать ссылку и присваивание значения *каждому* атрибуту, которые делаются в экземплярах. Во избежание зацикливания вставленные методы передавали бы допустимые запросы суперклассу с применением методик, обсуждаемых в предыдущей главе. Ниже показано, как выглядит потенциальное изменение кода нашего класса декоратора:

```
# Вставка методов: остальной код в access2.py остается прежним

def accessControl(failIf):
    def onDecorator(aClass):
        def getattributes(self, attr):
            trace('get:', attr)
            if failIf(attr):
                raise TypeError('private attribute fetch: ' + attr)
            else:
                return object.__getattribute__(self, attr)

        def setattributes(self, attr, value):
            trace('set:', attr)
            if failIf(attr):
                raise TypeError('private attribute change: ' + attr)
            else:
                return object.__setattr__(self, attr, value)

        aClass.__getattribute__ = getattributes
        aClass.__setattr__ = setattributes     # Вставка методов доступа
        return aClass                          # Возвращение исходного класса
    return onDecorator
```

Такая альтернативная версия решает проблему с проверкой типа, но привносит ряд других проблем. С одной стороны, этот декоратор может использоваться только клиентами в виде *классов нового стиля*: поскольку метод `__getattribute__` доступен только в классах нового стиля (как и `__setattr__`), декорированные классы в Python 2.X обязаны применять наследование нового стиля, что может быть или не быть подходящим для их целей. На самом деле набор поддерживаемых классов даже еще более ограничен: вставка методов нарушит работу клиентов, которые *уже используют* `__setattr__` или `__getattribute__` для собственных нужд.

Хуже того, представленная схема не решает проблему с атрибутами *встроенных* операций, описанную в предыдущем разделе, потому что метод `__getattribute__` тоже не запускается в таких контекстах. В нашем случае, если в `Person` есть метод `__str__`,

то он запустится операциями вывода, но лишь по причине его фактического наличия в классе. Как и ранее, атрибут `__str__` *не* будет направляться вставленному методу `__getattribute__` обобщенным образом – операция вывода просто проигнорирует данный метод и вызовет `__str__` класса напрямую.

Хотя вероятно поступать так лучше, чем вообще не поддерживать методы перегрузки операций в объекте-оболочке (за исключением минимум переопределения), данная схема по-прежнему не способна перехватывать и проверять доступ к методам `__X__`, что не позволяет делать любой из них закрытым. Должны ли методы перегрузки операций быть закрытыми – другой вопрос, но имеющаяся структура не допускает такой возможности.

Гораздо хуже то, что поскольку рассматриваемый подход без оболочки предусматривает добавление методов `__getattribute__` и `__setattr__` в декорированный класс, происходит перехват операций доступа к атрибутам, сделанным самим классом, и та же самая проверка допустимости, которая предпринимается в отношении операций доступа извне. Другими словами, собственные методы класса тоже не смогут пользоваться его закрытыми именами! Таким образом, подход со вставкой лишается каких-либо шансов быть примененным с пользой.

В действительности вставка этих методов подобным образом функционально эквивалентна их *наследованию* и влечет за собой те же самые ограничения, что и первоначальный код защиты из главы 30. Чтобы узнать, откуда инициирована операция доступа к атрибуту – изнутри или снаружи класса, нашим методам может понадобиться инспектирование объектов кадров в *стеке вызовов* Python. В конечном итоге может появиться решение – скажем, реализация закрытых атрибутов как свойств или дескрипторов, которые проверяют стек и проверяют допустимость только внешних операций доступа, – но доступ замедлился бы еще больше, а магии стало бы слишком много, чтобы исследовать здесь такой код. (Кажется, что дескрипторы делают возможным все, что угодно, даже когда не должны!)

Наряду с тем, что методика со вставкой методов интересна и вероятно уместна для ряда других сценариев использования, она не соответствует нашим целям. Мы не будем дополнительно исследовать такую кодовую схему в текущей главе, т.к. в следующей главе займемся изучением методик дополнения классов в сочетании с метаклассами. Там будет показано, что метаклассы не являются строго обязательными для изменения классов подобным образом, потому что декораторы классов часто способны исполнять ту же самую роль.

Python не поощряет контроль доступа

А теперь, когда потрачено столько усилий для реализации объявлений атрибутов `Private` и `Public` в коде Python, я обязан снова напомнить вам о том, что такое добавление в классы средств контроля доступа совершенно не соответствует стилю программирования на Python. На самом деле большинству программистов на Python данный пример покажется почти или вообще полностью неуместным, не считая демонстрации декораторов в действии. Большинство крупных программ на Python успешно обходятся вообще без средств контроля доступа подобного рода.

Однако вы можете счесть этот инструмент полезным в ограниченных рамках на стадии разработки. Если вы хотите привести в порядок доступ к атрибутам, чтобы устранить ошибки в коде, или недавно были программистом на C++ или Java, то знайте, что большинство вещей можно делать с помощью инструментов перегрузки операций и интроспекции Python.

Пример: проверка допустимости аргументов функций

В качестве финального примера, иллюстрирующего полезность декораторов, в настоящем разделе мы займемся созданием *декоратора функций*, который автоматически проверяет, находятся ли значения аргументов, переданных функции или методу, внутри допустимого числового диапазона. Он предназначен для применения на стадии разработки или в производственной среде и может использоваться как шаблон для решения похожих задач (скажем, проверка типов аргументов, когда она нужна). Из-за того, что ограничения на объем главы уже давно были превышены, код примера является материалом для самостоятельного изучения и сопровождается минимально необходимым описанием; как обычно, ищите дополнительные детали в файлах исходного кода.

Цель

В учебном руководстве по ООП в главе 28 мы написали класс, который обеспечивал повышение размера заработной платы для объектов, представляющих людей, на основе переданного процентного отношения:

```
class Person:
    ...
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Там было отмечено, что если нас интересует надежный код, то хорошей идеей будет выполнение проверки процентного отношения, чтобы оно не оказалось ни чересчур большим, ни слишком малым. Мы могли бы реализовать ее в самом методе с помощью операторов `if` или `assert` с применением *встраиваемых проверок*:

```
class Person:
    def giveRaise(self, percent):   # Проверка посредством встраиваемого кода
        if percent < 0.0 or percent > 1.0:
            raise TypeError, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))

class Person:                       # Проверка посредством утверждений
    def giveRaise(self, percent):
        assert percent >= 0.0 and percent <= 1.0, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))
```

Тем не менее, при таком подходе метод загромождается встраиваемыми проверками, которые вероятно будут полезными только на стадии разработки. В более сложных случаях их добавление может стать утомительным (представьте себе попытку встроить код, который необходим для реализации защиты доступа к атрибутам, обеспечиваемой декоратором из предыдущего раздела). Вероятно еще хуже то, что если когда-нибудь логику проверки понадобится изменить, то придется искать и обновлять множество встраиваемых копий в коде.

Более практичной и интересной альтернативой стала бы разработка универсального инструмента, способного автоматически выполнять проверки на предмет вхождения в диапазон для аргументов любой функции или метода, который мы можем написать сейчас или в будущем. Подход с *декоратором* делает это явным и удобным:

```
class Person:
    @rangetest(percent=(0.0, 1.0))     # Использование декоратора для проверки
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Изолирование логики проверки допустимости в декораторе упрощает как код клиентов, так и будущее сопровождение.

Обратите внимание, что наша текущая цель отличается от проверки допустимости атрибутов, реализованной в последнем примере предыдущей главы. Здесь мы подразумеваем проверку значений *аргументов функций* при их передаче, а не значений атрибутов, когда они устанавливаются. Инструменты декораторов и интроспекции Python позволяют нам решить эту новую задачу в равной степени легко.

Базовый декоратор проверки вхождения значений в диапазон для позиционных аргументов

Давайте начнем с реализации базовой проверки вхождения в диапазон. Чтобы излишне не усложнять, первым делом мы создадим декоратор, который работает только для позиционных аргументов и предполагает, что в каждом вызове они всегда занимают те же самые позиции. Их нельзя передавать по ключевым именам, и мы не поддерживаем в вызовах дополнительный аргумент `**args`, т.к. он может сделать недействительными позиции, объявленные в декораторе. Код находится в файле `rangetest1.py`:

```
def rangetest(*argchecks):    # Проверка вхождения в диапазон
                              #   позиционных аргументов
    def onDecorator(func):
        if not __debug__:   # True, если python -O main.py аргументы...
            return func     # Ничего не делать: вызвать исходную функцию напрямую
        else:               # Иначе определить оболочку на период отладки
            def onCall(*args):
                for (ix, low, high) in argchecks:
                    if args[ix] < low or args[ix] > high:
                        errmsg = 'Argument %s not in %s..%s' % (ix, low, high)
                                 # Аргумент не в диапазоне
                        raise TypeError(errmsg)
                return func(*args)
            return onCall
    return onDecorator
```

В имеющемся виде код главным образом представляет собою новую форму исследованных ранее кодовых схем: мы используем аргументы декоратора, вложенные области видимости для предохранения состояния и т.д.

Мы также применяем вложенные операторы `def`, чтобы обеспечить работу с простыми функциями и *методами*, как объяснялось ранее. При использовании в отношении метода класса `onCall` получает в первом элементе `*args` экземпляр целевого класса и передает его `self` в исходной функции метода; в этом случае номера аргументов в проверках на вхождение диапазона, начинаются с 1, а не с 0.

Новым здесь является применение в коде встроенной переменной `__debug__` — интерпретатор Python устанавливает ее в `True`, если только он не запускается с флагом командной строки `-O`, управляющим оптимизацией (например, `python -O main.py`). Когда значение `__debug__` равно `False`, декоратор возвращает исходную функцию без изменений, чтобы избежать последующих дополнительных вызовов и связанного

с ними снижения производительности. Другими словами, декоратор автоматически *удаляет* свою логику дополнения, когда используется -O, не требуя физического удаления декорирующих строк из кода.

Вот как применяется первое решение:

```
# Файл rangetest1_test.py
from __future__ import print_function     # Python 2.X
from rangetest1 import rangetest
print(__debug__)                    # False, если python -O main.py

@rangetest((1, 0, 120))             # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):            # Значение age должно быть в диапазоне 0..120
   print('%s is %s years old' % (name, age))

@rangetest([0, 1, 12], [1, 1, 31], [2, 0, 2009])
def birthday(M, D, Y):
   print('birthday = {0}/{1}/{2}'.format(M, D, Y))

class Person:
   def __init__(self, name, job, pay):
      self.job = job
      self.pay = pay

   @rangetest([1, 0.0, 1.0])  # giveRaise = rangetest(...)(giveRaise)
   def giveRaise(self, percent):  # Аргумент 0 здесь - экземпляр self
      self.pay = int(self.pay * (1 + percent))

# Закомментированные строки генерируют исключение TypeError,
# если только не используется командная строка python -O

persinfo('Bob Smith', 45)       # В действительности запускается onCall(...)
                                #   с состоянием
# persinfo('Bob Smith', 200)    # Или объект Person, если указан флаг -O

birthday(5, 31, 1963)
# birthday(5, 32, 1963)

sue = Person('Sue Jones', 'dev', 100000)
sue.giveRaise(.10)              # В действительности запускается onCall(self, .10)
print(sue.pay)                  # Или giveRaise(self, .10), если указан флаг -O
# sue.giveRaise(1.10)
# print(sue.pay)
```

При запуске допустимые вызовы в этом коде производят следующий вывод (весь код в настоящем разделе работает одинаково в Python 2.X и Python 3.X, потому что декораторы функций поддерживаются в обеих линейках, мы не используем делегирование атрибутов, а также применяем нейтральные к версии методики создания исключений и вывода):

```
C:\code> python rangetest1_test.py
True
Bob Smith is 45 years old
birthday = 5/31/1963
110000
```

Удаление комментария из любого недопустимого вызова приводит к генерации декоратором исключения `TypeError`. Ниже приведен результат, когда последним двум строкам разрешено выполняться (как обычно, ради экономии места часть текста сообщений об ошибках опущена):

```
C:\code> python rangetest1_test.py
True
Bob Smith is 45 years old
birthday = 5/31/1963
110000
TypeError: Argument 1 not in 0.0..1.0
Ошибка типа: значение аргумента 1 не находится в диапазоне 0.0..1.0
```

Запуск интерпретатора Python с флагом -O командной строки отключит проверку вхождения в диапазон, но также позволит избежать накладных расходов в плане производительности, связанных с уровнем оболочки — в итоге мы напрямую вызываем исходные недекорированные функции. Исходя из предположения, что декоратор является только инструментом для отладки, флаг -O можно использовать в целях оптимизации программы для работы в производственной среде:

```
C:\code> python -O rangetest1_test.py
False
Bob Smith is 45 years old
birthday = 5/31/1963
110000
231000
```

Обобщение для поддержки также ключевых аргументов и стандартных значений

В предыдущей версии иллюстрировались необходимые основы, но она довольно ограничена, т.к. поддерживает проверку допустимости только аргументов, передаваемых по позициям, и не проверяет ключевые аргументы (на самом деле в ней предполагается, что ключевые аргументы не передаются способом, который делает номера позиций аргументов некорректными). Вдобавок предыдущая версия ничего не предпринимает в отношении аргументов со стандартными значениями, которые могут не указываться в заданном вызове. Это нормально, если все аргументы передаются по позициям и никогда не имеют стандартных значений, но далеко от идеала в универсальном инструменте. В Python поддерживаются гораздо более гибкие режимы передачи аргументов, к которым мы пока не обращаемся.

Демонстрируемая далее видоизмененная версия нашего примера работает лучше. За счет сопоставления ожидаемых аргументов внутренней функции с фактическими аргументами, переданными в вызове, она поддерживает проверки вхождения в диапазон значений аргументов, которые передаются по позициям или по ключевым именам, и пропускает проверку аргументов со стандартными значениями, опущенными в вызове. Короче говоря, подлежащие проверке аргументы указываются ключевыми аргументами декоратора, который позже проходит по кортежу *pargs с позиционными аргументами и словарю **kargs с ключевыми аргументами для проверки допустимости их значений.

```
"""
Файл rangetest.py: декоратор функций, который выполняет проверку вхождения
в диапазон для аргументов, переданных любой функции или методу.

Аргументы указываются декоратору по ключам. В фактическом вызове аргументы
могут передаваться по позициям или по ключам, а аргументы со стандартными
значениями разрешено опускать. Примеры сценариев использования ищите в файле
rangetest_test.py.
"""
```

```
trace = True

def rangetest(**argchecks):   # Проверять вхождение в диапазон для аргументов
                              # обоих видов и аргументов со стандартными значениями
    def onDecorator(func):    # onCall запоминает func и argchecks
        if not __debug__:     # True, если python -O main.py аргументы...
            return func       # Поместить в оболочку в случае отладки,
                              # иначе использовать исходную функцию
        else:
            code = func.__code__
            allargs = code.co_varnames[:code.co_argcount]
            funcname = func.__name__

            def onCall(*pargs, **kargs):
                # Все элементы pargs соответствуют первым N
                #   ожидаемых аргументов по позициям
                # Остальные должны быть в kargs или быть опущены
                #   из-за наличия стандартных значений
                expected = list(allargs)
                positionals = expected[:len(pargs)]

                for (argname, (low, high)) in argchecks.items():
                    # Для всех аргументов, подлежащих проверке
                    if argname in kargs:
                        # Был передан по имени
                        if kargs[argname] < low or kargs[argname] > high:
                            # Аргумент не находится в диапазоне
                            errmsg = '{0} argument "{1}" not in {2}..{3}'
                            errmsg = errmsg.format(funcname, argname, low, high)
                            raise TypeError(errmsg)
                    elif argname in positionals:
                        # Был передан по позиции
                        position = positionals.index(argname)
                        if pargs[position] < low or pargs[position] > high:
                            # Аргумент не находится в диапазоне
                            errmsg = '{0} argument "{1}" not in {2}..{3}'
                            errmsg = errmsg.format(funcname, argname, low, high)
                            raise TypeError(errmsg)
                    else:
                   # Предполагается, что не был передан: имеет стандартное значение
                        if trace:
                            # Аргумент имеет стандартное значение
                            print('Argument "{0}" defaulted'.format(argname))

                return func(*pargs, **kargs) # Нормально: выполнить исходный вызов
            return onCall
    return onDecorator
```

В следующем тестовом сценарии показано, как применять декоратор – подлежащие проверке аргументы задаются ключевыми аргументами декоратора, а в реальных вызовах мы можем передавать аргументы по именам или по позициям и опускать аргументы со стандартными значениями, даже если они должны проверяться иным способом:

```
"""
Файл rangetest_test.py (Python 3.X + 2.X)
Закомментированные строки генерируют исключение TypeError,
если только не используется командная строка python -O
"""
```

```
from __future__ import print_function       # Python 2.X
from rangetest import rangetest

# Тестирование функций с позиционными и ключевыми аргументами

@rangetest(age=(0, 120))          # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):
    print('%s is %s years old' % (name, age))

@rangetest(M=(1, 12), D=(1, 31), Y=(0, 2013))
def birthday(M, D, Y):
    print('birthday = {0}/{1}/{2}'.format(M, D, Y))

persinfo('Bob', 40)
persinfo(age=40, name='Bob')
birthday(5, D=1, Y=1963)
# persinfo('Bob', 150)
# persinfo(age=150, name='Bob')
# birthday(5, D=40, Y=1963)

# Тестирование методов с позиционными и ключевыми аргументами

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay
                                      # giveRaise = rangetest(...)(giveRaise)
    @rangetest(percent=(0.0, 1.0))   # percent передается по имени или по позиции
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

bob = Person('Bob Smith', 'dev', 100000)
sue = Person('Sue Jones', 'dev', 100000)
bob.giveRaise(.10)
sue.giveRaise(percent=.20)
print(bob.pay, sue.pay)
# bob.giveRaise(1.10)
# bob.giveRaise(percent=1.20)

# Тестирование опущенных аргументов со стандартными значениями

@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))
def omitargs(a, b=7, c=8, d=9):
    print(a, b, c, d)

omitargs(1, 2, 3, 4)
omitargs(1, 2, 3)
omitargs(1, 2, 3, d=4)
omitargs(1, d=4)
omitargs(d=4, a=1)
omitargs(1, b=2, d=4)
omitargs(d=8, c=7, a=1)

# omitargs(1, 2, 3, 11)          # Недопустимое значение аргумента d
# omitargs(1, 2, 11)             # Недопустимое значение аргумента c
# omitargs(1, 2, 3, d=11)        # Недопустимое значение аргумента d
# omitargs(11, d=4)              # Недопустимое значение аргумента a
# omitargs(d=4, a=11)            # Недопустимое значение аргумента a
# omitargs(1, b=11, d=4)         # Недопустимое значение аргумента b
# omitargs(d=8, c=7, a=11)       # Недопустимое значение аргумента a
```

При выполнении сценария аргументы со значениями, выходящими за допустимые пределы, генерируют исключение, как и ранее, но аргументы можно передавать либо по именам, либо по позициям, а опущенные аргументы со стандартными значениями не проверяются. Код запускается под управлением Python 2.X и 3.X. Отследите вывод кода и поэкспериментируйте с кодом самостоятельно; он работает прежним образом, но его границы стали более широкими:

```
C:\code> python rangetest_test.py
Bob is 40 years old
Bob is 40 years old
birthday = 5/1/1963
110000 120000
1 2 3 4
Argument "d" defaulted
1 2 3 9
1 2 3 4
Argument "c" defaulted
Argument "b" defaulted
1 7 8 4
Argument "c" defaulted
Argument "b" defaulted
1 7 8 4
Argument "c" defaulted
1 2 8 4
Argument "b" defaulted
1 7 7 8
```

Когда мы удалим комментарий с одной из тестовых строк, сгенерируется исключение, связанное с ошибкой проверки допустимости, если только интерпретатору Python не был передан флаг -O командной строки, отключающий логику декоратора:

```
TypeError: giveRaise argument "percent" not in 0.0..1.0
TypeError: значение аргумента percent в giveRaise не находится в диапазоне
0.0..1.0
```

Детали реализации

Код декоратора rangetest полагается на API-интерфейс интроспекции и тонкие ограничения передачи аргументов. Для обеспечения большей универсальности в принципе мы могли бы попытаться воспроизвести логику сопоставления аргументов Python во всей ее полноте, чтобы видеть, какие имена в каких режимах были переданы, но такая сложность чрезмерна для нашего инструмента. Было бы лучше, если бы мы могли сопоставлять имена проверяемых аргументов, предоставленных декоратору, с именами реальных аргументов, ожидаемых функцией, с целью выяснения, как первые отображаются на вторые во время заданного вызова.

Интроспекция функций

Оказывается, что API-интерфейс интроспекции, доступный для объектов функций и ассоциированных с ними объектов кода, располагает в точности тем инструментом, который нам нужен. Данный API-интерфейс был кратко представлен в главе 19 первого тома, но фактически мы задействуем его только здесь. Набор имен ожидаемых аргументов — это просто первые *N* имен переменных, присоединенных к объекту кода функции:

```
# В Python 3.X (и Python 2.6+ для совместимости)
>>> def func(a, b, c, e=True, f=None):   # Аргументы: три обязательных,
                                          # два со стандартными значениями
        x = 1                             # Плюс еще две локальные переменные
        y = 2

>>> code = func.__code__                  # Объект кода объекта функции
>>> code.co_nlocals
7
>>> code.co_varnames                      # Все имена локальных переменных
('a', 'b', 'c', 'e', 'f', 'x', 'y')
>>> code.co_varnames[:code.co_argcount]   # <== Первые N имен локальных переменных
                                          # являются ожидаемыми аргументами
('a', 'b', 'c', 'e', 'f')
```

Как обычно, *помеченные звездочкой имена* в посреднике вызовов позволяет собрать произвольно много аргументов, подлежащих сопоставлению с ожидаемыми аргументами, которые получены из API-интерфейса интроспекции функции:

```
>>> def catcher(*pargs, **kargs): print('%s, %s' % (pargs, kargs))

>>> catcher(1, 2, 3, 4, 5)
(1, 2, 3, 4, 5), {}
>>> catcher(1, 2, c=3, d=4, e=5)          # Аргументы в вызовах
(1, 2), {'d': 4, 'e': 5, 'c': 3}
```

Такой API-интерфейс объекта функции доступен в более старых версиях Python, но в Python 2.5 и предшествующих версиях атрибут `func.__code__` назывался `func.func_code`; для совместимости новый атрибут `__code__` также избыточно доступен в Python 2.6 и последующих версиях. Дополнительные детали легко получить, вызвав `dir` на объектах функции и кода. Код следующего вида поддерживается в Python 2.5 и предшествующих версиях, хотя сам результат `sys.version_info` аналогично непереносим — в последних версиях Python он является именованным кортежем, но мы можем одинаково использовать смещения в более новых и старых версиях Python:

```
>>> import sys                            # Для обратной совместимости
>>> tuple(sys.version_info)               # [0] - старший номер версии
(3, 3, 0, 'final', 0)
>>> code = func.__code__ if sys.version_info[0] == 3 else func.func_code
```

Допущения относительно аргументов

Помимо набора имен ожидаемых аргументов декорированной функции решение опирается на два ограничения, которые интерпретатор Python накладывает на *порядок* передачи аргументов (они остаются в силе в текущих выпусках линеек Python 2.X и Python 3.X):

- в вызове все позиционные аргументы находятся перед всеми ключевыми аргументами;
- в определении `def` все аргументы без стандартных значений находятся перед всеми аргументами со стандартными значениями.

То есть в общем случае при *вызове* неключевой аргумент не может следовать за ключевым аргументом, а при *определении* аргумент без стандартного значения не может следовать за аргументом со стандартным значением. В обоих местах все синтаксические конструкции *имя=значение* обязаны находиться после любой конструкции вида просто *имя*. Как уже известно, интерпретатор Python сопоставляет значения аргумен-

тов, переданные по позициям, с именами аргументов в заголовках функций слева направо, так что эти значения всегда соответствуют *крайним слева* именам в заголовках. Напротив, ключевые аргументы сопоставляются по имени, и заданный аргумент может получить только одно значение.

Чтобы упростить работу, мы также можем сделать допущение о том, что в целом вызов *действителен*, т.е. все аргументы либо получат значения (по именам или по позициям), либо будут намеренно опущены для принятия стандартных значений. Такое допущение не обязательно будет соблюдаться, поскольку на момент выполнения проверки допустимости логикой оболочки функция еще не вызвана — вызов по-прежнему может потерпеть неудачу позже, когда инициируется уровнем оболочки, из-за некорректной передачи аргументов. Однако до тех пор, пока это не приводит к более серьезному отказу оболочки, мы можем обходить действительность вызова. Такой прием содействует упрощению кода, потому что проверка действительности вызовов до того, как они фактически совершаются, потребовала бы эмуляции алгоритма сопоставления аргументов Python в полной мере — чересчур сложной процедуры для нашего инструмента.

Алгоритм сопоставления

С учетом рассмотренных выше ограничений и допущений теперь мы можем разрешить указывать в вызове ключевые аргументы и опускать аргументы со стандартными значениями согласно данному алгоритму. Когда вызов перехвачен, мы в состоянии сделать следующие предположения и выводы.

1. Пусть *N* будет количеством переданных позиционных аргументов, полученным на основе длины кортежа `*pargs`.
2. Все *N* позиционных аргументов в `*pargs` должны соответствовать первым *N* ожидаемым аргументам, которые получены из объекта кода функции. Это вытекает из обрисованных ранее правил упорядочивания вызовов, т.к. все позиционные аргументы в вызове предшествуют всем ключевым аргументам.
3. Для получения имен аргументов, фактически переданных по позициям, мы можем извлечь из списка всех ожидаемых аргументов срез с длиной *N*, равной размеру кортежа `*pargs` с переданными позиционными аргументами.
4. Любые аргументы, расположенные после первых *N* ожидаемых аргументов, были либо переданы по ключевым именам, либо опущены в вызове и получили свои стандартные значения.
5. Для каждого имени аргумента, подлежащего проверке декоратором, мы должны выполнить указанные ниже действия.
   а) Если имя аргумента находится в `**kargs`, то аргумент передавался по имени — индексирование `**kargs` дает переданное значение.
   б) Если имя аргумента присутствует в первых *N* ожидаемых аргументах, то аргумент передавался по позиции — его относительная позиция в списке ожидаемых аргументов дает его относительную позицию в `*pargs`.
   в) В противном случае мы можем предположить, что аргумент был опущен в вызове и получил свое стандартное значение, а потому в проверке не нуждается.

Другими словами, мы можем пропускать проверки для аргументов, которые не указывались в вызове, предполагая то, что первые *N* фактически переданных позиционных аргументов в `*pargs` обязаны соответствовать первым *N* именам аргументов в списке всех ожидаемых аргументов, а также то, что остальные должны либо быть

переданы по ключевым словам и потому попасть в `**kargs`, либо получить свои стандартные значения. При такой схеме в целом декоратор просто не проверяет любой аргумент, который был опущен между крайним справа позиционным аргументом и крайним слева ключевым аргументом, между ключевыми аргументами или после крайнего справа позиционного аргумента. Отследите код декоратора и его тестовый сценарий, чтобы понять, как все реализовано.

Нерешенные проблемы

Несмотря на то что наш инструмент проверки на предмет вхождения в диапазон работает запланированным образом, остались нерешенными три проблемы — он не выявляет недействительные вызовы, не обрабатывает ряд сигнатур с произвольными аргументами и не полностью поддерживает вложение. Модернизация может потребовать расширения или совершенно других подходов. Ниже представлено краткое изложение проблем.

Недействительные вызовы

Во-первых, как упоминалось ранее, вызовы исходной функции, которые *не являются действительными*, все же терпят неудачу в нашем финальном декораторе. Скажем, оба следующих вызова генерируют исключения:

```
omitargs()
omitargs(d=8, c=7, b=6)
```

Тем не менее, они терпят неудачу, только когда мы пробуем обратиться к исходной функции в конце оболочки. Хотя мы могли бы сымитировать алгоритм сопоставления аргументов Python, чтобы избежать проблемы, поступать так не имеет особого смысла. Поскольку в указанной точке вызов в любом случае откажет, мы также может позволить собственной логике сопоставления аргументов Python самостоятельно обнаруживать проблему.

Произвольные аргументы

Во-вторых, несмотря на то, что наша финальная версия декоратора обрабатывает позиционные аргументы, ключевые аргументы и опущенные аргументы со стандартными значениями, она по-прежнему не предпринимает никаких явных действий в отношении имен аргументов со звездочками *`*pargs`* и *`**kargs`*, которые могут применяться в декорированной функции, принимающей *произвольно много* аргументов. Однако для имеющихся целей нам, вероятно, не следует об этом беспокоиться.

- Если передается дополнительный ключевой аргумент, то его имя появится в `**kargs` и может быть нормально проверено, если оно указано в декораторе.
- Если дополнительный ключевой аргумент не передается, то его имя будет отсутствовать в `**kargs` или в созданном с помощью среза списке ожидаемых позиционных аргументов, и потому он не проверяется. Он обрабатывается, как если бы с ним было связано стандартное значение, хотя в действительности это необязательный дополнительный аргумент.
- Если передается дополнительный позиционный аргумент, тогда в любом случае отсутствует какой-либо способ сослаться на него в декораторе — его имя будет отсутствовать в `**kargs` и в созданном с помощью среза списке ожидаемых аргументов, а потому он просто пропускается. Поскольку такие аргументы не указаны в определении функции, то отобразить имя, предоставленное декоратору, на относительную позицию ожидаемого аргумента невозможно.

Другими словами, код поддерживает проверку произвольных ключевых аргументов по именам, но не произвольных позиционных аргументов, которые не имеют имени и, следовательно, установленной позиции в сигнатуре функции. Что касается API-интерфейса объекта функции, то вот эффект использования таких инструментов в декорированных функциях:

```
>>> def func(*kargs, **pargs): pass
>>> code = func.__code__
>>> code.co_nlocals, code.co_varnames
(2, ('kargs', 'pargs'))
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(0, ())

>>> def func(a, b, *kargs, **pargs): pass
>>> code = func.__code__
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(2, ('a', 'b'))
```

Из-за того, что имена аргументов со звездочками проявляют себя как локальные переменные, но *не* как ожидаемые аргументы, они не участвуют в нашем алгоритме сопоставления – имена, предшествующие им в заголовках функций, могут проверяться обычным образом, но не любые переданные дополнительные позиционные аргументы. В принципе мы могли бы расширить интерфейс декоратора, чтобы поддерживать в декорированной функции также и **pargs* для тех редких случаев, когда это оказывается полезным (например, для специального имени аргумента с проверкой, применяемой ко всем аргументам в **pargs* оболочки, которые расположены в позициях с номерами, превышающими длину списка ожидаемых аргументов), но рассматривать здесь такое расширение мы не будем.

Вложение декораторов

В-третьих, что возможно будет самым тонким моментом, задействованный в коде подход не полностью поддерживает использование *вложения декораторов* для комбинирования шагов. Поскольку он анализирует аргументы с применением имен в определениях функций, а имена в функции-посреднике вызовов, возвращаемой вложенным декорированием, не соответствуют именам аргументов ни в исходной функции, ни в аргументах декоратора, поддержка его использования во вложенном режиме оказывается неполной.

Формально при вложении полностью выполняется только самая глубоко вложенная проверка допустимости; на всех остальных уровнях вложенности проверки запускаются только для аргументов, переданных по ключевым словам. Отследите код, чтобы выяснить причину; из-за того, что сигнатура вызова оболочки `onCall` не ожидает именованных позиционных аргументов, любые подлежащие проверке аргументы, переданные по позициям, трактуются так, как если бы они были опущены и получили стандартные значения, а потому попросту пропускаются.

Это может быть неотъемлемой частью принятого в инструменте подхода – посредники изменяют сигнатуры с именами аргументов на своих уровнях, делая невозможным прямое отображение имен в аргументах декораторов на позиции в последовательностях переданных аргументов. При наличии посредников *имена* аргументов в итоге применяются только к ключевым словам; наоборот, *позиции* аргументов в первом пробном решении могут лучше поддерживать посредников, но не полностью поддерживать ключевые слова.

Вместо того, чтобы наделять декоратор способностью к вложению, мы обобщим его для поддержки многих видов проверки допустимости в единственном декорировании при ответе на контрольный вопрос главы, где также приведем примеры ограничений вложения. Тем не менее, поскольку мы почти исчерпали объем книги, запланированный под настоящий пример, дальнейшие усовершенствования оставляются в качестве упражнения для самостоятельной проработки.

Аргументы декоратора или аннотации функций

Интересно отметить, что аннотации функций, введенные в Python 3.X (в Python 3.0 и последующих версиях) могли бы предложить альтернативу аргументам декоратора, используемым в примере для указания проверок вхождения в диапазон. Как обсуждалось в главе 19 первого тома, аннотации позволяют ассоциировать выражения с аргументами и возвращаемыми значениями, записывая их в самой строке заголовка def; интерпретатор Python накапливает аннотации в словаре и присоединяет его к аннотированной функции.

В нашем примере мы могли бы вместо аргументов декоратора применять аннотации функций, чтобы закодировать границы диапазона в строке заголовка. Нам все еще требовался бы декоратор функции для помещения функции в оболочку с целью перехвата последующих вызовов, но по существу мы обменяли бы следующий синтаксис аргументов декоратора:

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                    # func = rangetest(...)(func)
    print(a + b + c)
```

на синтаксис аннотаций такого вида:

```
@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)
```

То есть ограничения по диапазону были бы перенесены внутрь самой функции вместо того, чтобы записываться внешне. В приведенном далее сценарии иллюстрируется структура результирующих декораторов для обеих схем с сокращенным скелетным кодом ради краткости. Кодовая схема с аргументами декоратора соответствует нашему полному решению, показанному ранее; альтернативная версия с аннотациями требует на один уровень меньше вложения, т.к. не нуждается в предохранении аргументов декоратора в качестве состояния:

```
# Использование аргументов декоратора (Python 3.X + 2.X)

def rangetest(**argchecks):
    def onDecorator(func):
        def onCall(*pargs, **kargs):
            print(argchecks)
            for check in argchecks:
                pass                  # Добавить сюда код проверки допустимости
            return func(*pargs, **kargs)
        return onCall
    return onDecorator

@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                    # func = rangetest(...)(func)
    print(a + b + c)

func(1, 2, c=3)              # Запускается onCall, argchecks в области видимости
```

```
# Использование аннотаций функций (только Python 3.X)

def rangetest(func):
    def onCall(*pargs, **kargs):
        argchecks = func.__annotations__
        print(argchecks)
        for check in argchecks:
            pass                      # Добавить сюда код проверки допустимости
        return func(*pargs, **kargs)
    return onCall

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)): # func = rangetest(func)
    print(a + b + c)

func(1, 2, c=3)                       # Запускается onCall, аннотации на функции
```

При запуске обе схемы имеют доступ к той же самой информации, касающейся проверки допустимости, но в разных формах. Для версии с аргументами декоратора информация хранится внутри аргумента в объемлющей области видимости, а для версии с аннотациями — в атрибуте самой функции. Вот результат использования аннотаций функций в Python 3.X:

```
C:\code> py -3 decoargs-vs-annotation.py
{'a': (1, 5), 'c': (0.0, 1.0)}
6
{'a': (1, 5), 'c': (0.0, 1.0)}
6
```

Я оставляю реализацию остальной части версии, основанной на аннотациях, в качестве упражнения для самостоятельной проработки; его код будет идентичен показанному ранее полному решению, потому что информация, касающаяся проверки вхождения в диапазон, просто связывается с функцией, а не с объемлющей областью видимости. На самом деле мы лишь приобретаем другой пользовательский интерфейс для нашего инструмента — как и ранее, для получения относительных позиций по-прежнему необходимо сопоставлять имена аргументов с именами ожидаемых аргументов.

Фактически применение аннотаций вместо аргументов декоратора в этом примере, как ни странно, *ограничивает его полезность*. С одной стороны, аннотации работают только в Python 3.X, так что Python 2.X больше не поддерживается; с другой стороны, декораторы функций с аргументами работают в обеих линейках.

Что более важно, за счет перемещения спецификаций для проверки допустимости в заголовок def мы по существу фиксируем функцию на *единственной роли* — т.к. аннотации позволяют записывать только одно выражение на аргумент, они могут иметь лишь одно целевое назначение. Скажем, мы не можем использовать аннотации проверки вхождения в диапазон в любой другой роли.

Наоборот, поскольку аргументы декоратора записываются вне самой функции, они легче в удалении и *более универсальны* — код самой функции вовсе не подразумевает единственную цель декорирования. Критически важно то, что за счет *вложения* декораторов с аргументами мы можем применять множество шагов дополнения к той же самой функции; аннотация напрямую поддерживает только один шаг. Благодаря аргументам декораторов сами функции также сохраняют более простой и нормальный внешний вид.

Однако если вы преследуете единственную цель и можете поддерживать только Python 3.X, тогда выбор между аннотациями и аргументами декораторов по большей части будет лишь стилистическим и субъективным. Как часто бывает в жизни, деко-

рирование или аннотации одного человека вполне могут оказаться синтаксическим беспорядком для другого!

Другие приложения: проверка типов (если вы настаиваете!)

Кодовую схему, к которой мы пришли при обработке аргументов в декораторах, можно было бы применить в других контекстах. Например, прямолинейным расширением является проверка типов данных аргументов:

```
def typetest(**argchecks):
    def onDecorator(func):
        ...
        def onCall(*pargs, **kargs):
            positionals = list(allargs)[:len(pargs)]
            for (argname, type) in argchecks.items():
                if argname in kargs:
                    if not isinstance(kargs[argname], type):
                        ...
                        raise TypeError(errmsg)
                elif argname in positionals:
                    position = positionals.index(argname)
                    if not isinstance(pargs[position], type):
                        ...
                        raise TypeError(errmsg)
                else:
                    # Предположить, что не передавался: стандартное значение
            return func(*pargs, **kargs)
        return onCall
    return onDecorator

@typetest(a=int, c=float)
def func(a, b, c, d):                    # func = typetest(...)(func)
    ...

func(1, 2, 3.0, 4)                       # Нормально
func('spam', 2, 99, 4)                   # Корректно генерирует исключение
```

Использование аннотаций функции вместо аргументов декоратора для такого декоратора, как был описан в предыдущем разделе, сделало бы внешний вид кода еще более похожим на объявление типов на других языках:

```
@typetest
def func(a: int, b, c: float, d):  # func = typetest(func)
    ...                            # Вот как!...
```

Но здесь мы подобрались опасно близко к тому, чтобы перестать играть по правилам. Как вы должны были узнать из этой книги, такая конкретная роль обычно является плохой идеей для воплощения в рабочем коде и подобно закрытым объявлениям совершенно не соответствует стилю программирования на Python (а также часто симптомом бывшего программиста на C++, который пытается перейти на Python).

Проверка типов сужает возможности вашей функции до работы только с конкретными типами вместо того, чтобы позволить ей оперировать с любыми типами, имеющими совместимые *интерфейсы*. В действительности она ограничивает ваш код и снижает его *гибкость*. С другой стороны, из каждого правила есть исключение; проверка типов может оказаться полезной в отдельных случаях на стадии отладки и при взаимодействии с кодом, написанным на более ограничивающих языках, таких как C++.

Тем не менее, эта универсальная схема обработки аргументов может быть применима и в различных менее спорных ролях. Мы могли бы продолжить обобщение, передавая *проверочную функцию*, почти как при добавлении объявлений `Public` ранее; затем единственной копии кода такого вида было бы достаточно для проверки на предмет вхождения в диапазон и проверки типов, а возможно и для других похожих ролей. На самом деле мы *будем* делать подобное обобщение при ответе на контрольные вопросы в конце главы, а пока оставляем данное расширение как предстоящее увлекательное приключение.

Резюме

В главе мы исследовали декораторы — их разновидности для функций и классов. Как выяснилось, декораторы предоставляют способ вставки кода, который должен автоматически выполняться при определении функции или класса. Когда используется декоратор, интерпретатор Python повторно привязывает имя функции или класса к вызываемому объекту, возвращенному декоратором. Такая привязка позволяет управлять самими функциями и классами или последующими обращениями к ним — за счет добавления уровня с логикой оболочки для перехвата будущих вызовов мы можем дополнять вызовы функций и интерфейсы экземпляров. Как было также показано, управляющие функции и ручная повторная привязка позволяют добиться той же цели, но декораторы обеспечивают более явное и единообразное решение.

Кроме того, декораторы классов можно применять для управления самим классами, а не только их экземплярами. Поскольку такая функциональность перекрывается с функциональностью *метаклассов* (тема следующей главы книги), вы должны продолжить чтение, чтобы ознакомиться с окончанием этой истории и в целом книги. Однако сначала постарайтесь ответить на контрольные вопросы текущей главы. Так как основное внимание в главе было сосредоточено на реализации примеров, вам будет предложено модифицировать их код. Вы найдете код исходных версий в пакете примеров, доступном для загрузки. В случае нехватки времени изучите модификации, предложенные в ответах — программирование предусматривает не только написание кода, но и его чтение.

Проверьте свои знания: контрольные вопросы

1. *Декораторы методов.* Как упоминалось в одной из врезок "На заметку!" ранее в главе, декоратор функций для измерения времени, принимающий аргументы (из модуля `timerdeco2.py`), который мы реализовали в разделе "Добавление аргументов к декоратору", может применяться только к простым *функциям*, поскольку для перехвата вызовов он использует вложенный класс с методом перегрузки операций `__call__`. Такая структура не работает для *методов* класса, потому что в `self` передается экземпляр декоратора, а не экземпляр целевого класса.

   Перепишите этот декоратор так, чтобы его можно было применять к простым функциям и методам в классах, и протестируйте его на функциях и методах. (Подсказка: ищите рекомендации в разделе "Грубые ошибки, связанные с классами, часть I: декорирование методов" ранее в главе.) Обратите внимание, что для отслеживания суммарного времени вам, вероятно, понадобится использовать атрибуты объекта функции, т.к. вы не будете иметь вложенный класс для предохранения состояния и не сможете получать доступ к нелокальным пере-

менным извне кода декоратора. В качестве бонуса декоратор станет пригодным к употреблению в Python 3.X и 2.X.

2. *Декораторы классов.* Декораторы классов `Public/Private` из модуля `access2.py`, которые были написаны в первом учебном примере главы, добавляют *накладные расходы в плане производительности* к каждой операции извлечения атрибута в декорированном классе. Хотя мы могли бы просто удалить строку с декорированием `@`, чтобы ускорить выполнение, можно также дополнить сам декоратор проверкой встроенной переменной `__debug__` и не помещать класс в оболочку, когда в командной строке интерпретатора Python передается флаг `-O` (именно так мы делали для декораторов, проверяющих вхождение в диапазон). Это позволит ускорять выполнение программы, не изменяя ее исходный код, через аргументы командной строки (`python -O main.py...`). Одновременно можно было бы также воспользоваться одной из изученных методик подмешиваемых суперклассов для перехвата нескольких *встроенных операций* в Python 3.X. Реализуйте и протестируйте указанные два расширения.

3. *Обобщенная проверка допустимости аргументов.* Декоратор функций и методов, написанный в `rangetest.py`, проверяет значения переданных аргументов на предмет вхождения в допустимый диапазон, но мы также выяснили, что ту же схему можно было бы применять для достижения похожих целей, таких как проверка типов аргументов и возможно что-то еще. Обобщите декоратор так, чтобы его единственную кодовую базу можно было использовать для множества проверок допустимости аргументов. С учетом имеющейся кодовой структуры простейшим решением могут оказаться передаваемые функции, хотя в контекстах, больше направленных на ООП, подклассы, которые предоставляют ожидаемые методы, также часто способны обеспечить аналогичные способы обобщения.

Проверьте свои знания: ответы

1. Ниже демонстрируется один способ решения для ответа на первый контрольный вопрос вместе с его выводом (хотя некоторые методы могут выполняться слишком быстро). Хитрость заключается в замене вложенных классов *вложенными функциями*, чтобы аргумент `self` не являлся экземпляром декоратора, и присваивание суммарного времени атрибуту самой декораторной функции для его последующего извлечения через повторно привязанное имя (детали ищите в разделе "Варианты предохранения состояния для декораторов" этой главы — функции поддерживают присоединение произвольных атрибутов, а имя функции в данном контексте представляет собой ссылку из объемлющей области видимости). Если вы желаете продолжить расширение решения, то в дополнение к суммарному времени можете также запоминать *наилучшее* (минимальное) время выполнения вызова, как делалось в примерах измерения времени в главе 21 первого тома.

```
"""
Файл timerdeco.py (Python 3.X + 2.X)
Декоратор для измерения времени выполнения вызовов для функций и методов.
"""
import time

def timer(label='', trace=True):       # При наличии аргументов декоратора:
                                       #   сохранение аргументов
```

```
    def onDecorator(func):              # При декорировании @: сохранение
                                        #   декорированной функции
        def onCall(*args, **kargs):     # При вызовах: вызов исходной функции
            start = time.clock()        # Состояние в области видимости +
                                        #   атрибут функции
            result = func(*args, **kargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            if trace:
                format = '%s%s: %.5f, %.5f'
                values = (label, func.__name__, elapsed, onCall.alltime)
                print(format % values)
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator
```

Тесты реализованы в отдельном файле, чтобы декоратор можно было легко использовать многократно:

```
"""
Файл timerdeco-test.py
"""
from __future__ import print_function       # Python 2.X
from timerdeco import timer
import sys
force = list if sys.version_info[0] == 3 else (lambda X: X)

print('------------------------------------------------------')
# Тестирование на функциях

@timer(trace=True, label='[CCC]==>')
def listcomp(N):                         # Подобно listcomp = timer(...)(listcomp)
    return [x * 2 for x in range(N)]    # listcomp(...) запускает onCall

@timer('[MMM]==>')
def mapcall(N):
    return force(map((lambda x: x * 2), range(N)))  # list() для представ-
                                                     #   лений Python 3.X

for func in (listcomp, mapcall):
    result = func(5)                    # Время для этого вызова, всех
                                        #   вызовов и возвращаемое значение
    func(5000000)
    print(result)
    print('allTime = %s\n' % func.alltime) # Суммарное время для всех вызовов

print('------------------------------------------------------')
# Тестирование на методах

class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @timer()
    def giveRaise(self, percent):       # giveRaise = timer()(giveRaise)
        self.pay *= (1.0 + percent)     # Декоратор запоминает giveRaise

    @timer(label='**')
    def lastName(self):                 # lastName = timer(...)(lastName)
```

```
        return self.name.split()[-1]        # alltime для каждого класса,
                                            #   не для экземпляра

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
bob.giveRaise(.10)
sue.giveRaise(.20)                          # Запускает onCall(sue, .10)
print(int(bob.pay), int(sue.pay))
print(bob.lastName(), sue.lastName())       # Запускает onCall(bob),
                                            #   запоминает lastName
print('%.5f %.5f' % (Person.giveRaise.alltime, Person.lastName.alltime))
```

Если все идет согласно плану, тогда вы увидите в Python 3.X и 2.X следующий вывод, хотя результаты измерения времени будут варьироваться в зависимости от версии Python и компьютера:

```
c:\code> py -3 timerdeco-test.py
--------------------------------------------------
[CCC]==>listcomp: 0.00001, 0.00001
[CCC]==>listcomp: 0.57930, 0.57930
[0, 2, 4, 6, 8]
allTime = 0.5793010457092784

[MMM]==>mapcall: 0.00002, 0.00002
[MMM]==>mapcall: 1.08609, 1.08611
[0, 2, 4, 6, 8]
allTime = 1.0861149923442373
--------------------------------------------------
giveRaise: 0.00001, 0.00001
giveRaise: 0.00000, 0.00001
55000 120000
**lastName: 0.00001, 0.00001
**lastName: 0.00000, 0.00001
Smith Jones
0.00001 0.00001
```

2. Следующие три файла дают ответ на второй контрольный вопрос. В первом файле находится *декоратор* — он был дополнен для возвращения исходного класса в режиме оптимизации (-O), так что операции доступа к атрибутам не приводят к снижению скорости. По большей части здесь были добавлены операторы проверки, активен ли режим отладки, и дополнительные отступы вправо для кода класса:

```
"""
Файл access.py (Python 3.X + 2.X)
Декоратор классов с объявлениями атрибутов Private и Public.
Управляет внешним доступом к атрибутам, которые хранятся
в экземпляре или унаследованы из классов любым способом.

Private объявляет имена атрибутов, которые нельзя извлекать
или присваивать снаружи декорированного класса, а Public
объявляет все имена, для которых это разрешено.

Предостережение: в Python 3.X встроенные операции перехватываются
только в BuiltinMixins (требует расширения); в имеющемся виде Public
для перегрузки операций может быть менее полезным, чем Private.
"""
```

```
from access_builtins import BuiltinsMixin    # Частичный набор!

traceMe = False
def trace(*args):
   if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
   def onDecorator(aClass):
      if not __debug__:
         return aClass
      else:
         class onInstance(BuiltinsMixin):
            def __init__(self, *args, **kargs):
               self.__wrapped = aClass(*args, **kargs)

            def __getattr__(self, attr):
               trace('get:', attr)
               if failIf(attr):
                  raise TypeError('private attribute fetch: ' + attr)
               else:
                  return getattr(self.__wrapped, attr)

            def __setattr__(self, attr, value):
               trace('set:', attr, value)
               if attr == '_onInstance__wrapped':
                  self.__dict__[attr] = value
               elif failIf(attr):
                  raise TypeError('private attribute change: ' + attr)
               else:
                  setattr(self.__wrapped, attr, value)
         return onInstance
   return onDecorator

def Private(*attributes):
   return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
   return accessControl(failIf=(lambda attr: attr not in attributes))
```

Я также использовал одну из методик подмешивания для переопределения в классе оболочки нескольких методов перегрузки операций, так что в Python 3.X он корректно делегирует выполнение встроенных операций целевым классам, которые используют данные методы. В том виде, как есть, в Python 2.X посредник является стандартным классическим классом, который уже прогоняет их через `__getattr__`, но в Python 3.X посредник представляет собой класс нового стиля, который так не поступает. Применяемый здесь подмешиваемый класс требует перечисления таких методов в декораторах `Public`; просмотрите приводимые ранее альтернативы, которые этого не требуют (но также не разрешают делать встроенные операции закрытыми), и при необходимости расширьте класс:

```
"""
Файл access_builtins.py (из access2_builtins2b.py)
Направляет некоторые встроенные операции обратно методу __getattr__
класса-посредника, так что они работают в Python 3.X аналогично прямым
вызовам по именам и стандартным классическим классам Python 2.X.
При необходимости расширьте, чтобы включить другие
имена методов __X__, используемые внутренними объектами.
"""
```

```
class BuiltinsMixin:
   def reroute(self, attr, *args, **kargs):
      return self.__class__.__getattr__(self, attr)(*args, **kargs)

   def __add__(self, other):
      return self.reroute('__add__', other)
   def __str__(self):
      return self.reroute('__str__')
   def __getitem__(self, index):
      return self.reroute('__getitem__', index)
   def __call__(self, *args, **kargs):
      return self.reroute('__call__', *args, **kargs)

   # Плюс любые другие, используемые внутренними объектами, в Python 3.X
```

Здесь я снова вынес код самотестирования в отдельный файл, чтобы декоратор можно было импортировать в других местах, не запуская тесты, а также для того, чтобы не проверять `__name__` и не делать отступы:

```
"""
Файл: access-test.py
Код тестов: в отдельном файле, чтобы сделать возможным многократное
использование декоратора.
"""
import sys
from access import Private, Public

print('---------------------------------------------------------')
# Тест 1: имена открыты, если не объявлены закрытыми

@Private('age')                  # Person = Private('age')(Person)
class Person:                    # Person = onInstance с состоянием
   def __init__(self, name, age):
      self.name = name
      self.age = age             # Внутренние операции доступа
                                 #   выполняются нормально

   def __add__(self, N):
      self.age += N              # Встроенные операции перехватываются
                                 # подмешиваемым классом в Python 3.X

   def __str__(self):
      return '%s: %s' % (self.name, self.age)

X = Person('Bob', 40)
print(X.name)                    # Внешние операции доступа проверяются
X.name = 'Sue'
print(X.name)
X + 10
print(X)

try: t = X.age                   # ТЕРПИТ НЕУДАЧУ, если только
                                 #   не используется python -O
except: print(sys.exc_info()[1])
try: X.age = 999                 # То же самое
except: print(sys.exc_info()[1])

print('---------------------------------------------------------')
# Тест 2: имена закрыты, если не объявлены открытыми
# Операции должны быть не Private или Public, если используются в BuiltinMixin
```

```
@Public('name', '__add__', '__str__', '__coerce__')
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age
   def __add__(self, N):
      self.age += N                    # Встроенные операции перехватываются
                                       # подмешиваемым классом в Python 3.X
   def __str__(self):
      return '%s: %s' % (self.name, self.age)

X = Person('bob', 40)                  # X - это onInstance
print(X.name)                          # onInstance содержит Person
X.name = 'sue'
print(X.name)
X + 10
print(X)

try: t = X.age                         # ТЕРПИТ НЕУДАЧУ, если только не
используется python -O
except: print(sys.exc_info()[1])
try: X.age = 999                       # То же самое
except: print(sys.exc_info()[1])
```

Наконец, если все работает ожидаемо, тогда вывод выглядит следующим образом в Python 3.X и 2.X — тот же самый код применяется к тому же самому классу, декорированному с помощью `Private` и затем `Public`:

```
c:\code> py -3 access-test.py
------------------------------------------------------------
Bob
Sue
Sue: 50
private attribute fetch: age
private attribute change: age
------------------------------------------------------------
bob
sue
sue: 50
private attribute fetch: age
private attribute change: age

c:\code> py -3 -O access-test.py       # Четыре сообщения об ошибках доступа
                                       #   не показаны
```

3. Ниже приведена обобщенная версия декоратора для самостоятельного изучения. Она использует передаваемую функцию проверки, которой предоставляются критерии проверки, указанные для аргумента в декораторе. Поддерживаются проверки на предмет вхождения в диапазон, проверки типов, проверки значений и почти все, что только можно придумать в таком выразительном языке, как Python. Я также провел небольшой рефакторинг кода для устранения избыточности и автоматизировал обработку неудачных проверок. Примеры применения и ожидаемый вывод представлены в коде самотестирования модуля. Согласно описанному ранее предостережению в том виде, как есть, этот декоратор не полностью работает во вложенном режиме (для позиционных аргументов выполняется только наиболее глубоко вложенная проверка), но произвольную функцию

valuetest можно использовать для комбинирования разных видов проверок при одном декорировании (хотя объем кода, необходимого в данном режиме, может свести на нет большинство его преимуществ перед простым assert!).

```
"""
Файл argtest.py (Python 3.X + 2.X): декоратор функций, который выполняет
произвольные переданные проверки для аргументов, передаваемых любой функции
или методу. Двумя примерами могут служить проверка на предмет вхождения
в диапазон и проверка типов; valuetest обрабатывает более произвольные
проверки для значения аргумента.

Аргументы для декоратора указываются по ключевым словам. В фактическом
вызове аргументы могут передаваться по позициям или по ключевым словам,
а аргументы со стандартными значениями разрешено опускать. Примеры
сценариев использования приведены в коде самотестирования.

Предостережение: не полностью поддерживает вложение, потому что аргументы
посредника вызовов отличаются; не проверяет дополнительные аргументы,
переданные в *args декорированного класса; может оказаться не проще,
чем assert, исключая заготовленные сценарии использования.
"""
trace = False

def rangetest(**argchecks):
    return argtest(argchecks, lambda arg, vals: arg < vals[0] or arg >
vals[1])

def typetest(**argchecks):
    return argtest(argchecks, lambda arg, type: not isinstance(arg, type))

def valuetest(**argchecks):
    return argtest(argchecks, lambda arg, tester: not tester(arg))

def argtest(argchecks, failif):  # Проверка аргументов согласно failif
                                 #   и критерия
    def onDecorator(func):       # onCall предохраняет func, argchecks, failif
        if not __debug__:        # Ничего не делать, если python -O main.py
                                 #   аргументы...
            return func
        else:
            code = func.__code__
            expected = list(code.co_varnames[:code.co_argcount])
            def onError(argname, criteria):
                errfmt = '%s argument "%s" not %s'
                raise TypeError(errfmt % (func.__name__, argname, criteria))

            def onCall(*pargs, **kargs):
                positionals = expected[:len(pargs)]
                for (argname, criteria) in argchecks.items(): # Для всего,
                                                              # что проверяется
                    if argname in kargs:               # Передается по имени
                        if failif(kargs[argname], criteria):
                            onError(argname, criteria)

                    elif argname in positionals:       # Передается по позиции
                        position = positionals.index(argname)
                        if failif(pargs[position], criteria):
                            onError(argname, criteria)
                    else:            # Опущен, т.к. имеет стандартное значение
                        if trace:
```

```
                    print('Argument "%s" defaulted' % argname)
            return func(*pargs, **kargs)      # Нормально: запустить
                                              #   исходный вызов
        return onCall
    return onDecorator

if __name__ == '__main__':
    import sys
    def fails(test):
        try: result = test()
        except: print('[%s]' % sys.exc_info()[1])
        else: print('?%s?' % result)

    print('--------------------------------------------------------------')
    # Заготовленные сценарии использования: диапазоны, типы

    @rangetest(m=(1, 12), d=(1, 31), y=(1900, 2013))
    def date(m, d, y):
        print('date = %s/%s/%s' % (m, d, y))

    date(1, 2, 1960)
    fails(lambda: date(1, 2, 3))

    @typetest(a=int, c=float)
    def sum(a, b, c, d):
        print(a + b + c + d)

    sum(1, 2, 3.0, 4)
    sum(1, d=4, b=2, c=3.0)
    fails(lambda: sum('spam', 2, 99, 4))
    fails(lambda: sum(1, d=4, b=2, c=99))

    print('--------------------------------------------------------------')
    # Произвольные/смешанные проверки

    @valuetest(word1=str.islower, word2=(lambda x: x[0].isupper()))
    def msg(word1='mighty', word2='Larch', label='The'):
        print('%s %s %s' % (label, word1, word2))

    msg()                          # word1 и word2 имеют стандартные значения
    msg('majestic', 'Moose')
    fails(lambda: msg('Giant', 'Redwood'))
    fails(lambda: msg('great', word2='elm'))

    print('--------------------------------------------------------------')
    # Ручные проверки типов и вхождения в диапазон

    @valuetest(A=lambda x: isinstance(x, int), B=lambda x: x > 0 and x < 10)
    def manual(A, B):
        print(A + B)

    manual(100, 2)
    fails(lambda: manual(1.99, 2))
    fails(lambda: manual(100, 20))

    print('--------------------------------------------------------------')
    # Вложение: выполняются обе проверки за счет вложения посредников
    # для исходной функции.
    # Нерешенная проблема: внешние уровни не проверяют позиционные аргументы
    # из-за отличающейся сигнатуры функции посредника вызовов;
    # когда trace=True, вследствие сигнатуры посредника все аргументы X
    # кроме последнего классифицируются как имеющие стандартные значения.
```

```
@rangetest(X=(1, 10))
@typetest(Z=str)                 # Только самый внутренний уровень
                                 #   проверяет позиционные аргументы
def nester(X, Y, Z):
    return('%s-%s-%s' % (X, Y, Z))

print(nester(1, 2, 'spam'))              # Исходная функция выполняется
                                         #   надлежащим образом
fails(lambda: nester(1, 2, 3))           # Вложенный typetest
                                         #   выполняется: позиционные
fails(lambda: nester(1, 2, Z=3))         # Вложенный typetest
                                         #   выполняется: ключевой
fails(lambda: nester(0, 2, 'spam'))      # <== Внешний rangetest не
                                         #   выполняется: позиционные
fails(lambda: nester(X=0, Y=2, Z='spam')) # Внешний rangetest выполняется:
                                          #   ключевые
```

Ниже приведен вывод кода самотестирования модуля в Python 3.X и 2.X (некоторые объекты Python 2.X отображаются слегка иначе): как обычно, лучше понять вывод поможет изучение исходного кода.

```
c:\code> py -3 argtest.py
--------------------------------------------------------------------
date = 1/2/1960
[date argument "y" not (1900, 2013)]
10.0
10.0
[sum argument "a" not <class 'int'>]
[sum argument "c" not <class 'float'>]
--------------------------------------------------------------------
The mighty Larch
The majestic Moose
[msg argument "word1" not <method 'islower' of 'str' objects>]
[msg argument "word2" not <function <lambda> at 0x0000000002A096A8>]
--------------------------------------------------------------------
102
[manual argument "A" not <function <lambda> at 0x0000000002A09950>]
[manual argument "B" not <function <lambda> at 0x0000000002A09B70>]
--------------------------------------------------------------------
1-2-spam
[nester argument "Z" not <class 'str'>]
[nester argument "Z" not <class 'str'>]
?0-2-spam?
[onCall argument "X" not (1, 10)]
```

И напоследок, как мы уже знаем, кодовая структура этого декоратора работает с функциями и методами:

```
# Файл argtest_testmeth.py
from argtest import rangetest, typetest

class C:
    @rangetest(a=(1, 10))
    def meth1(self, a):
        return a * 1000
  @typetest(a=int)
  def meth2(self, a):
      return a * 1000
```

```
>>> from argtest_testmeth import C
>>> X = C()
>>> X.meth1(5)
5000
>>> X.meth1(20)
TypeError: meth1 argument "a" not (1, 10)
Ошибка типа: аргумент a в meth1 не (1, 10)
>>> X.meth2(20)
20000
>>> X.meth2(20.9)
TypeError: meth2 argument "a" not <class 'int'>
Ошибка типа: аргумент a в meth2 не <class 'int'>
```

ГЛАВА 40

Метаклассы

В предыдущей главе мы исследовали декораторы и ознакомились с разнообразными примерами их использования. В этой главе мы продолжим уделять внимание построению инструментов и изучим еще одну сложную тему: *метаклассы.*

Метаклассы в некотором смысле просто расширяют модель вставки кода, поддерживаемую декораторами. Как выяснилось в предыдущей главе, декораторы функций и классов позволяют перехватывать и дополнять вызовы функций и вызовы, создающие экземпляры классов. В том же духе метаклассы дают возможность перехватывать и дополнять *создание классов* — они предоставляют API-интерфейс для вставки дополнительной логики, подлежащей выполнению в завершение оператора `class`, хотя и не такими способами, как декораторы. Соответственно, они обеспечивают универсальный протокол для управления объектами классов в программе.

Подобно всему, что рассматривается в текущей части книги, метаклассы относятся к *сложным темам*, которые можно исследовать по мере необходимости. На практике метаклассы позволяют получить высокий уровень контроля над тем, как работает набор классов. Это мощная концепция, и метаклассы не предназначены для большинства прикладных программистов. Откровенно говоря, тема метаклассов также и не для слабонервных — некоторые части главы могут требовать особого внимания (а иные вполне можно было бы приписать перу Доктора Сьюза (`https://ru.wikipedia.org/wiki/Доктор_Сьюз`)!).

С другой стороны, метаклассы открывают двери для различных кодовых схем, возможно трудных или неосуществимых по-другому. Они особенно интересны программистам, которые стремятся реализовывать гибкие *API-интерфейсы* либо инструменты программирования для применения другими. Однако даже если вы не входите в указанную категорию, метаклассы могут многое вам рассказать о модели классов Python в целом (как будет показано, они влияют также на *наследование*), а их знание будет необходимым условием для понимания кода, где они задействованы. Подобно прочим продвинутым инструментам метаклассы начали появляться в программах на Python чаще, чем рассчитывали их создатели.

Как и в предыдущей главе, часть нашей цели заключается в том, чтобы также продемонстрировать более реалистичные примеры кода, нежели показанные ранее в книге. Хотя метаклассы относятся к основному языку и сами по себе не образуют отдельную предметную область, в главе планируется пробудить у вас интерес к исследованию более крупных примеров программирования прикладных приложений по окончании чтения настоящей книги.

Поскольку в книге это последняя глава с техническим содержанием, в ней также завершается ряд цепочек обсуждений, касающихся самого языка Python, с которы-

ми мы часто сталкивались ранее и которые будут доведены до конца в последующем заключении. Разумеется, то, чем вы займетесь после чтения данной книги, зависит лишь от вас, но в проекте с открытым кодом важно помнить общую картину, занимаясь мелкими деталями.

Нужно ли иметь дело с метаклассами?

Возможно, метаклассы являются самой сложной темой в этой книге, а то и во всем языке Python. Ниже приведено мнение Тима Петерса, заслуженного разработчика основной части языка Python (и также автора известного девиза в `import this`), которое было высказано им в группе новостей `comp.lang.python`.

[Метаклассы] — это более темная магия, которая не касается 99% пользователей. Если вы размышляете, нужны ли они вам, то наверняка нет (люди, которым действительно необходимы метаклассы, совершенно точно знают, что они нужны, и им не требуется объяснение причин).

Другими словами, метаклассы предназначены главным образом для подгруппы программистов, занимающихся построением API-интерфейсов и инструментов, которые будут использоваться другими. Во многих случаях (если только не в большинстве) они, скорее всего, не будут лучшим вариантом в работе приложений, что особенно справедливо, когда вы пишете код, который впоследствии будет эксплуатироваться другими людьми. Реализация чего-нибудь лишь потому, что оно выглядит "крутым", обычно не служит разумным оправданием, разве только если вы экспериментируете или учитесь.

Тем не менее, метаклассы имеют широкий спектр потенциальных сценариев применения и важно знать, когда они полезны. Например, они могут использоваться для расширения классов средствами вроде отслеживания, постоянства объектов, регистрации исключений и многого другого. С помощью метаклассов также можно конструировать порции класса во время выполнения на основе конфигурационных файлов, применять декораторы функций к каждому методу класса обобщенным образом, проверять соответствие ожидаемым интерфейсам и т.д.

В своих более грандиозных воплощениях метаклассы можно даже использовать при реализации альтернативных кодовых схем, таких как аспектно-ориентированное программирование, инструменты объектно-реляционного отображения для баз данных и т.п. Хотя часто существуют другие способы достижения таких результатов (как выяснится, роли *декораторов классов* и метаклассов нередко пересекаются), метаклассы предоставляют формальную модель, приспособленную для таких задач. Конечно, нам не хватит места в этой главе, чтобы исследовать все приложения подобного рода, но после изучения основ рекомендуется поискать дополнительные сценарии применения в веб-сети.

Вероятно, наиболее подходящая причина исследования метаклассов в настоящей книге заключается в том, что они помогают прояснить механизм классов Python в общем. Скажем, мы увидим, что метаклассы являются неотъемлемой частью модели наследования нового стиля в языке, которая здесь окончательно формализуется. Независимо от того, будете вы реализовывать либо использовать их в своей работе или нет, даже поверхностное знание метаклассов способно содействовать более глубокому пониманию языка Python в целом [1].

[1] Здесь уместно привести сообщение об ошибке, выдаваемое Python 3.X: "TypeError: metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases" ("Ошибка типа: конфликт метаклассов: метакласс производ-

ного класса должен быть (нестрогим) подклассом метаклассов всех его базовых классов"). Оно отражает ошибочное использование модуля как суперкласса, но метаклассы могут оказаться не настолько необязательными, как предполагали разработчики — мы вернемся к данной теме при подведении итогов по книге в следующей главе.

Повышение уровней "магии"

Основное внимание в книге было сосредоточено на методиках реализации приложений — модулях, функциях и классах, на написание которых большинство программистов тратят свое время, достигая стоящих перед ними целей. Большая часть пользователей Python могут взаимодействовать с классами, создавать экземпляры и даже иногда перегружать операции, но вряд ли они будут слишком глубоко вникать в детали того, как действительно работают их классы.

Однако в этой книге также рассматривались разнообразные инструменты, которые позволяют управлять поведением Python обобщенным образом и часто имеют больше отношения к внутренним механизмам Python или построению инструментов, чем к области прикладного программирования. Ниже приведен обзор, который поможет понять место, занимаемое метаклассами.

Атрибуты и инструменты интроспекции

Специальные атрибуты, такие как `__class__` и `__dict__`, позволяют просматривать внутренние детали реализации объектов Python с целью их обобщенной обработки — для получения списка всех атрибутов объекта, отображения имени класса и т.д. Как мы увидим, инструменты наподобие `dir` и `getattr` способны исполнять похожие роли, когда должны поддерживаться "виртуальные" атрибуты вроде слотов.

Методы перегрузки операций

Особым образом именованные методы, такие как `__str__` и `__add__`, реализованные в классах, перехватывают и снабжают линиями поведения встроенные операции, которые применяются к экземплярам классов, включая вывод, операции выражений и т.д. Они запускаются автоматически в ответ на встроенные операции и предоставляют классам возможность соответствовать ожидаемым интерфейсам.

Методы перехвата доступа к атрибутам

Специальная категория методов перегрузки операций предлагает способ перехвата доступа к атрибутам экземпляров обобщенным образом: `__getattr__`, `__setattr__`, `__delattr__` и `__getattribute__` позволяют классам оболочек (т.е. посредникам) вставлять автоматически запускаемый код, который может проверять допустимость запросов атрибутов и делегировать их внутренним объектам. Они делают возможным выполнение вычислений любого количества атрибутов при доступе — либо избранных атрибутов, либо вообще всех.

Свойства классов

Встроенная функция `property` позволяет ассоциировать с конкретным атрибутом код, который автоматически запускается, когда происходит извлечение, присваивание или удаление атрибута. Несмотря на меньшую универсальность по сравнению с описанными выше инструментами, свойства делают возможным автоматическое выполнение кода при доступе к специфическим атрибутам.

Дескрипторы атрибутов классов

Вообще говоря, `property` является лаконичным путем определения дескриптора атрибута, который автоматически вызывает функции при доступе. Дескрипторы позволяют реализовывать в отдельном классе методы-обработчики `__get__`, `__set__` и `__delete__`, запускаемые автоматически, когда происходит доступ к атрибуту, которому присвоен экземпляр этого класса. Они обеспечивают универсальный способ вставки произвольного кода, который неявно выполняется при доступе к конкретному атрибуту как часть нормальной процедуры нахождения атрибутов.

Декораторы функций и классов

Как было показано в главе 39, особый синтаксис вида `@вызваемый_объект` для декораторов позволяет добавлять логику, подлежащую автоматическому запуску, когда происходит вызов функции или создание экземпляра класса. Такая логика оболочки может отслеживать либо измерять время работы вызовов, проверять допустимость значений аргументов, управлять всеми экземплярами класса, дополнять экземпляры добавочной линией поведения вроде проверки допустимости операций извлечения атрибутов и т.д. Декораторный синтаксис вставляет логику повторной привязки имен, которая должна выполняться в конце операторов определения функций и классов — имена декорированных функций и классов могут повторно привязываться либо к дополненным исходным объектам, либо к объектам-посредникам, которые перехватывают последующие обращения.

Метаклассы

Последний раздел магии, представленной в главе 32, который мы обсудим здесь.

Как упоминалось во введении данной главы, *метаклассы* являются продолжением рассматриваемой истории — они позволяют вставлять логику, подлежащую автоматическому выполнению в конце оператора `class`, когда создается объект класса. Хотя механизм метаклассов сильно напоминает декораторы классов, он не привязывает имя класса к результату, возвращенному вызываемым объектом декоратора, а взамен поручает процедуру *создания самого класса* специализированной логике.

Язык привязок

Другими словами, метаклассы в конечном итоге представляют собой лишь еще один способ определения *автоматически запускаемого кода*. Благодаря инструментам, перечисленным в предыдущем разделе, Python дает нам возможность вставлять логику в разнообразные контексты, среди которых выполнение операций, доступ к атрибутам, вызовы функций, создание экземпляров классов и теперь создание объектов классов. Это язык с *изобилием привязок* — средство, подобно любому другому открытое для злоупотреблений, но также обеспечивающее гибкость, о которой мечтает ряд программистов, и которая может требоваться в определенных программах.

Как было показано, со многими из расширенных инструментов Python связаны *пересекающиеся роли*. Например, атрибутами часто можно управлять с помощью свойств, дескрипторов или методов перехвата доступа к атрибутам. Далее в главе вы увидите, что декораторы классов и метаклассы также нередко могут использоваться взаимозаменяемо. В качестве краткого предварительного обзора:

- хотя декораторы классов часто применяются для управления экземплярами, они также могут использоваться для управления классами почти как метаклассы;
- аналогично наряду с тем, что метаклассы предназначены для дополнения процедуры создания классов, они также могут вставлять посредников для управления экземплярами почти как декораторы классов.

На самом деле главное функциональное отличие между указанными двумя инструментами касается момента, когда они активизируются в рамках *процесса* создания классов. В предыдущей главе объяснялось, что декораторы классов запускаются *после* того, как декорированный класс был создан. Таким образом, они часто применяются для добавления логики, подлежащей выполнению во время создания *экземпляров*. Когда декораторы классов снабжают класс линией поведения, то обычно делают это через изменения или посредников, а не более непосредственное отношение.

Как вы увидите здесь, по контрасту с декораторами классов метаклассы запускаются *во время* создания классов, чтобы создать и возвратить новый клиентский класс. Следовательно, они часто используются для управления или дополнения самих *классов* и могут даже предоставлять методы для обработки созданных из них классов через прямое отношение между экземплярами.

Скажем, метаклассы можно применять для автоматического декорирования всех методов классов, регистрации всех используемых классов в API-интерфейсе, автоматического добавления к классам пользовательского интерфейса, создания или расширения классов на основе упрощенных спецификаций в текстовых файлах и т.д. Поскольку они способны управлять тем, как создаются классы, и через посредников поведением, которым обзаводятся их экземпляры, применимость метаклассов потенциально очень широка.

Тем не менее, в главе будет показано, что во многих распространенных ролях эти два инструмента скорее похожи, чем различаются. Так как выбор инструментов для работы иногда отчасти субъективен, знание альтернатив может помочь подобрать правильный инструмент для решения имеющейся задачи. Чтобы лучше понять варианты, давайте посмотрим, что нам могут предложить метаклассы.

Недостаток "вспомогательных" функций

Также подобно декораторам из предыдущей главы с теоретической точки зрения метаклассы часто необязательны. Обычно мы можем достичь того же самого эффекта, пропуская объекты классов через *управляющие функции* (временами называемые *вспомогательными* функциями), что во многом похоже на возможность реализации целей декораторов за счет прогона функций и экземпляров через управляющий код. Однако, как и декораторы, метаклассы:

- обеспечивают более формальную и явную структуру;
- помогают гарантировать, что прикладные программисты не забудут дополнить свои классы в соответствии с требованиями API-интерфейса;
- снижают избыточность кода и связанные с ней накладные расходы на сопровождение, вынося логику настройки классов в единственное место, т.е. метакласс.

Для иллюстрации предположим, что нам нужно автоматически вставлять метод в набор классов. Разумеется, мы могли бы решить задачу посредством обычного *наследования*, если целевой метод известен на момент написания кода классов. В таком случае мы можем просто реализовать метод в суперклассе и заставить все необходимые классы наследоваться от суперкласса:

```
class Extras:
   def extra(self, args):      # Нормальное наследование: слишком статично
      ...
class Client1(Extras): ...     # Клиенты наследуют дополнительный метод extra
class Client2(Extras): ...
class Client3(Extras): ...

X = Client1()                  # Создание экземпляра
X.extra()                      # Запуск дополнительного метода extra
```

Тем не менее, иногда при написании кода классов спрогнозировать такое дополнение невозможно. Возьмем сценарий, когда классы дополняются в ответ на выбор, сделанный в пользовательском интерфейсе во время выполнения, или согласно спецификациям, введенным в конфигурационном файле. Несмотря на то что мы могли бы предусмотреть в каждом классе из нашего воображаемого набора код для *ручной* проверки этого, у клиентов придется выяснять множество вопросов (вызовы `required` здесь абстрактны и должны быть чем-то заполнены):

```
def extra(self, arg): ...

class Client1: ...             # Дополнения клиентов: крайне разбросаны
if required():
   Client1.extra = extra

class Client2: ...
if required():
   Client2.extra = extra

class Client3: ...
if required():
   Client3.extra = extra

X = Client1()
X.extra()
```

Мы можем добавлять методы в класс после оператора `class` из-за того, что метод уровня класса является всего лишь функцией, которая ассоциирована с классом и имеет первый аргумент, предназначенный для приема экземпляра `self`. Хотя прием работает, он может стать непригодным для более крупных наборов методов и возлагает все бремя дополнения на клиентские классы (вдобавок допуская, что они не забудут об этом!).

С точки зрения сопровождения было бы лучше изолировать логику выбора в одном месте. Мы могли бы инкапсулировать часть дополнительной работы, пропуская классы через *управляющую функцию*, которая бы должным образом расширяла класс и выполняла всю работу по проверке и конфигурированию во время выполнения:

```
def extra(self, arg): ...

def extras(Class):          # Управляющая функция: слишком много ручной работы
   if required():
      Class.extra = extra
class Client1: ...
extras(Client1)

class Client2: ...
extras(Client2)

class Client3: ...
extras(Client3)

X = Client1()
X.extra()
```

В коде класс прогоняется через управляющую функцию немедленно после создания. Несмотря на то что управляющие функции подобного рода позволяют здесь достичь нашей цели, они по-прежнему будут тяжелой ношей для разработчиков классов, которые обязаны понимать требования и придерживаться их в своем коде. Было бы лучше, если бы существовал простой способ принудительно навязать дополнение в целевых классах, чтобы разработчикам не приходилось иметь дело с дополнением настолько явно, и стало бы меньше шансов вообще о нем забыть. Другими словами, мы бы хотели иметь возможность вставлять какой-то код, подлежащий *автоматическому* запуску в конце оператора `class` для дополнения класса.

Именно такую работу выполняют *метаклассы* – объявляя метакласс, мы сообщаем интерпретатору Python о том, что процесс создания объекта класса должен быть направлен указанному нами другому классу:

```
def extra(self, arg): ...

class Extras(type):
    def __init__(Class, classname, superclasses, attributedict):
        if required():
            Class.extra = extra

class Client1(metaclass=Extras): ...    # Только объявление метакласса
                                         #   (форма Python 3.X)
class Client2(metaclass=Extras): ...    # Клиентский класс является
                                         #   экземпляром метакласса
class Client3(metaclass=Extras): ...

X = Client1()                            # X - экземпляр Client1
X.extra()
```

Поскольку интерпретатор Python автоматически активизирует метакласс в конце оператора `class`, когда новый класс создан, он может необходимым образом дополнять, регистрировать или по-другому управлять классом. Кроме того, единственное требование для клиентских классов заключается в том, что они объявляют метакласс; каждый класс, который так поступает, автоматически получит любое дополнение, предоставляемое метаклассом, и теперь, и в будущем, если метакласс изменится.

Конечно, приведенное обоснование выглядит стандартным и вам придется выработать собственное суждение – действительно, разработчики клиентских классов в той же степени же легко могут забыть об указании метакласса, как и вызывать управляющую функцию! Однако явная природа метаклассов может уменьшить вероятность такого забывания. Более того, метаклассы обладают дополнительными возможностями, которые пока еще не раскрывались. Хотя в таком небольшом примере это трудно заметить, метаклассы обычно лучше справляются с задачами подобного рода по сравнению с более ручными подходами.

Метаклассы против декораторов классов: раунд 1

Важно также отметить, что *декораторы классов*, описанные в предыдущей главе, иногда пересекаются с метаклассами – с точки зрения полезности и выгоды. Несмотря на то что декораторы классов часто используются для управления экземплярами, они также могут дополнять классы независимо от любых созданных экземпляров. Синтаксис делает их применение аналогично явным и возможно более очевидным, чем вызовы управляющих функций.

Скажем, пусть мы реализовали управляющую функцию, которая возвращает дополненный класс вместо того, чтобы модифицировать его на месте. Это позволило бы

достичь более высокой степени гибкости, т.к. управляющая функция могла бы возвращать любой тип объекта, который реализует ожидаемый интерфейс класса:

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

class Client1: ...
Client1 = extras(Client1)

class Client2: ...
Client2 = extras(Client2)

class Client3: ...
Client3 = extras(Client3)

X = Client1()
X.extra()
```

Если вы думаете, что код начинает напоминать декораторы классов, то абсолютно правы. В предыдущей главе мы делали особый акцент на роли декораторов классов в дополнении вызовов, создающих *экземпляры*. Тем не менее, из-за того, что они работают, выполняя автоматическую повторную привязку имени класса к результату функции, нет никаких причин, по которым мы не могли бы их использовать для дополнения класса, изменяя его перед тем, как будут созданы любые экземпляры. То есть декораторы классов могут применять дополнительную логику к *классам*, а не только к *экземплярам*, во время создания классов:

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

@extras
class Client1: ...    # Client1 = extras(Client1)

@extras
class Client2: ...    # Повторная привязка класса независимо от экземпляров

@extras
class Client3: ...

X = Client1()         # Создание экземпляра дополненного класса
X.extra()             # X - экземпляр исходного класса Client1
```

Здесь декораторы по существу автоматизируют повторное привязывание имен из предыдущего примера. Как и для метаклассов, поскольку декоратор возвращает исходный класс, экземпляры создаются из него, а не из объекта-оболочки. Фактически в приведенном примере создание экземпляров вообще не перехватывается.

В продемонстрированном особом случае — добавление методов к классу, когда он создается — выбор между метаклассами и декораторами весьма произволен. Декораторы могут использоваться для управления экземплярами и классами, и сильнее всего они пересекаются с метаклассами во второй из указанных ролей, но такое различение не абсолютно. На самом деле роли каждого инструмента частично определяются их механизмами.

Как вы увидите далее, декораторы формально соответствуют методам `__init__` метаклассов, применяемым для инициализации вновь созданных классов. Однако помимо инициализации классов метаклассы имеют дополнительные привязки для настройки и способны выполнять произвольные задачи создания классов, решение которых с помощью декораторов может оказаться более сложным. В итоге они могут стать сложнее, но и будут лучше подходить для дополнения классов по мере их формирования.

Например, в метаклассах также есть метод `__new__`, используемый для создания класса, который не имеет аналога в декораторах; создание нового класса в декораторе потребовало бы дополнительного шага. Кроме того, метаклассы могут предоставлять линии поведения, получаемые классами в форме *методов*, что тоже не имеет прямого эквивалента в декораторах; декораторы обязаны снабжать класс поведением менее прямыми способами.

И наоборот, поскольку метаклассы предназначены для управления классами, их использование для управления только *экземплярами* не настолько оптимально. Ввиду того, что они также несут ответственность за создание самого класса, в ролях управления экземплярами возникает *добавочный* шаг.

Позже в главе мы исследуем отличия в коде и доведем неполный код из текущего раздела до реалистичного рабочего примера. Тем не менее, чтобы понять работу метаклассов, сначала необходимо получить более четкое представление о лежащей в их основе модели.

Магия тут и магия там

Список в разделе "Повышение уровней 'магии'" ранее в главе охватывал разновидности магии за рамками тех, которые у программистов принято считать полезными. Кто-то может добавить к этому списку инструменты Python для *функционального* программирования, такие как замыкания и генераторы, и даже базовую поддержку ООП. Первые опираются на предохранение области видимости и автоматическое создание генераторных объектов, а ООП – на поиск атрибутов при наследовании и особый первый аргумент в функциях. Хотя они тоже базируются на магии, но представляют парадигмы, которые облегчают задачу программирования, предлагая абстракцию поверх лежащей в основе аппаратной архитектуры.

Например, поддержка ООП – более ранняя парадигма Python – широко принята в мире разработки программного обеспечения. Она предоставляет модель для написания программ, которая является более полной, явной и богато структурированной, чем инструменты функционального программирования. То есть некоторые уровни магии считаются более обоснованными, нежели другие; в конце концов, если бы не толика магии, то программы все еще состояли бы из машинного кода (или физических коммутаторов).

Как правило, *накопление* новой магии подвергает системы риску превысить порог сложности – скажем, добавление парадигмы функционального программирования к языку, который всегда был объектно-ориентированным, либо введение избыточности или замысловатых способов достижения целей, которые редко преследуются на практике большинством пользователей. Такая магия способна поднять планку слишком высоко для основной части аудитории вашего инструмента.

Кроме того, одна магия навязывается своим пользователям сильнее другой. Например, шаг трансляции компилятора обычно не требует от пользователей быть разработчиками компилятора. И наоборот, встроенная функция Python по имени `super` предполагает наличие полноценного мастерства и ввода в действие возможно неясного и неестественного алгоритма MRO. Представленный в этой главе алгоритм

наследования нового стиля похожим образом считает обязательным знание дескрипторов, метаклассов и MRO – самих по себе сложных инструментов. Даже неявные "привязки" вроде дескрипторов остаются неявными лишь до первого их отказа или цикла сопровождения. *Откровенная магия* подобного рода усиливает предварительные условия, требуемые инструментом, и снижает удобство его использования.

В системах с открытым кодом только время и количество загрузок способны определить, где могут находиться пороги сложности. Отыскание надлежащего *баланса* между мощностью и сложностью зависит от переменчивого мнения в такой же степени, как и от технологии. Однако, оставив в стороне субъективные факторы, навязываемая пользователям новая магия неизбежно делает более крутой кривую обучения для системы – тема, к которой мы вернемся в заключительных словах финальной главы.

Модель метаклассов

Для понимания метаклассов сначала необходимо чуть больше узнать о модели типов Python и о том, что происходит в конце оператора `class`. Как вы увидите, они тесно взаимосвязаны.

Классы являются экземплярами `type`

До сих пор в книге мы выполняли б*о*льшую часть работы, создавая экземпляры встроенных типов вроде списков и строк, а также экземпляры классов, которые реализовывали самостоятельно. Вы видели, что экземпляры *классов* не только имеют ряд собственных атрибутов информации состояния, но и наследуют поведенческие атрибуты от классов, из которых они были созданы. То же самое остается справедливым для *встроенных* типов; например, экземпляры списка располагают собственными значениями и вдобавок наследуют методы из спискового типа.

Хотя мы можем многое делать с такими объектами экземпляров, модель типов Python оказывается несколько богаче, чем было формально описано. По правде говоря, в показанной до сих пор модели имеется пробел: если экземпляры создаются из классов, тогда что создает наши *классы*? Выясняется, что классы тоже являются экземплярами чего-то:

- в Python 3.X объекты классов, определяемых пользователей, представляют собой экземпляры объекта по имени `type`, который сам по себе является классом.
- в Python 2.X классы нового стиля наследуются от `object`, который представляет собой подкласс `type`; классические классы являются экземплярами `type` и не создаются из какого-то класса.

Мы исследовали понятие типов в главе 9 первого тома, а взаимоотношение между классами и типами в главе 32, но давайте здесь проанализируем основы, чтобы увидеть, как они применяются к метаклассам.

Вспомните, что встроенная функция `type` возвращает тип любого объекта (который тоже объект), когда вызывается с единственным аргументом. Для встроенных типов наподобие списков типом экземпляра будет встроенный списковый тип, но типом спискового типа оказывается сам `type` – объект `type` на верхушке иерархии создает индивидуальные типы, а индивидуальные типы создают экземпляры. Вы можете убедиться в этом самостоятельно в интерактивной оболочке. Скажем, в Python 3.X типом экземпляра списка является списковый класс, типом которого будет класс `type`:

```
C:\code> py -3                          # В Python 3.X:
>>> type([]), type(type([]))            # Экземпляр списка создается
                                        #   из спискового класса
(<class 'list'>, <class 'type'>)        # Списковый класс создается из класса type
>>> type(list), type(type)              # То же самое, но с именами типов
(<class 'type'>, <class 'type'>)        # Типом type является type: верхушка иерархии
```

Как выяснилось в главе 32 при изучении изменений, внесенных в классы нового стиля, то же самое в целом справедливо в Python 2.X, но типы не полностью совпадают с классами. Здесь `type` представляет собой уникальный вид встроенного объекта, который покрывает иерархию типов и используется для создания типов:

```
C:\code> py -2
>>> type([]), type(type([]))            # В Python 2.X тип type немного отличается
(<type 'list'>, <type 'type'>)
>>> type(list), type(type)
(<type 'type'>, <type 'type'>)
```

Между прочим, отношение типы/экземпляры сохраняется также для определяемых пользователем классов: экземпляры создаются из классов, а классы создаются из `type`. Тем не менее, в Python 3.X понятие "типа" объединено с понятием "класса". На самом деле по существу это два синонима: *классы являются типами, а типы – классами.* То есть:

- типы определяются классами, которые унаследованы от `type`;
- определяемые пользователем классы являются экземплярами класса `type`;
- определяемые пользователем классы представляют собой типы, которые генерируют собственные экземпляры.

Ранее было показано, что такая эквивалентность влияет на код, который проверяет тип экземпляров: типом экземпляра будет класс, из которого экземпляр был сгенерирован. Она также имеет значение для способа, которым классы создаются, что оказывается важным моментом для понимания темы текущей главы. Поскольку по умолчанию классы обычно создаются из корневого класса `type`, большинству программистов не приходится думать об эквивалентности типов и классов. Однако это открывает новые возможности по настройке классов и их экземпляров.

Например, все определяемые пользователем классы в Python 3.X (и классы нового стиля в Python 2.X) являются экземплярами класса `type`, а объекты экземпляров – экземплярами своих классов. В действительности классы теперь имеют атрибут `__class__`, ссылающийся на `type`, точно так же, как экземпляр имеет атрибут `__class__`, ссылающийся на класс, из которого он был создан:

```
C:\code> py -3
>>> class C: pass                       # Объект класса Python 3.X (нового стиля)
>>> X = C()                             # Объект экземпляра класса

>>> type(X)                             # Экземпляр является экземпляром класса
<class '__main__.C'>
>>> X.__class__                         # Класс экземпляра
<class '__main__.C'>

>>> type(C)                             # Класс является экземпляром type
<class 'type'>
>>> C.__class__                         # Классом класса является type
<class 'type'>
```

Обратите особое внимание на последние две строки – классы представляют собой экземпляры класса type, в точности как нормальные экземпляры являются экземплярами класса, определяемого пользователем. В Python 3.X это работает одинаково для встроенных типов и типов классов, определяемых пользователем. На самом деле классы вообще не считаются отдельной концепций: они представляют собой просто определяемые пользователем типы и сам type определен посредством класса.

В Python 2.X ситуация аналогична для классов нового стиля, производных от object, потому что тогда становится доступным поведение классов Python 3.X (как вы уже видели, Python 3.X автоматически добавляет object в кортеж суперклассов __bases__ корневых классов верхнего уровня, чтобы квалифицировать их как классы нового стиля):

```
C:\code> py -2
>>> class C(object): pass          # Классы нового стиля в Python 2.X,
>>> X = C()                        # классы тоже имеют атрибут __class__

>>> type(X)
<class '__main__.C'>
>>> X.__class__
<class '__main__.C'>

>>> type(C)
<type 'type'>
>>> C.__class__
<type 'type'>
```

Тем не менее, классические классы в Python 2.X несколько отличаются. Из-за того, что они отражают первоначальную модель классов из более старых версий Python, классические классы не имеют ссылки __class__ и подобно встроенным типам Python 2.X являются экземплярами объекта type, а не класса type (некоторые шестнадцатеричные адреса сокращены):

```
C:\code> py -2
>>> class C: pass                  # Классические классы в Python 2.X,
>>> X = C()                        # сами классы не имеют атрибута __class__

>>> type(X)
<type 'instance'>
>>> X.__class__
<class __main__.C at 0x005F85A0>

>>> type(C)
<type 'classobj'>
>>> C.__class__
AttributeError: class C has no attribute '__class__'
Ошибка атрибута: класс С не имеет атрибута __class__
```

Метаклассы являются подклассами type

Почему нас должен волновать тот факт, что классы являются экземплярами класса type в Python 3.X? Оказывается, что это привязка, которая предоставляет нам возможность реализации метаклассов. Поскольку в настоящее время понятия *типа* и *класса* совпадают, мы можем создавать подклассы type для его настройки с помощью обычных методик ООП и синтаксиса классов. А из-за того, что классы в действительности представляют собой экземпляры класса type, создание классов на основе настроенных подклассов type позволяет реализовывать специальные виды классов.

Обращаясь к деталям, все работает вполне естественно – в Python 3.X и в классах нового стиля Python 2.X:

- type является классом, который генерирует классы, определяемые пользователем;
- метаклассы представляют собой подклассы класса type;
- объекты классов являются экземплярами класса type или какого-то из его подклассов;
- объекты экземпляров генерируются из класса.

Другими словами, для управления способом создания классов и дополнения их поведения нам необходимо лишь указать, что определяемый пользователем класс должен создаваться из определяемого пользователем метакласса, а не нормального класса type.

Обратите внимание, что такое отношение между *типом* и *экземпляром* не вполне соответствует обычному наследованию. Определяемые пользователем классы могут также иметь суперклассы, из которых они и их экземпляры наследуют атрибуты. Как вы уже знаете, наследуемые суперклассы перечисляются внутри круглых скобок в операторе class и появляются в кортеже __bases__ класса. Однако с типом, из которого создается класс, и экземпляром которого он является, имеется другое отношение. Процедура наследования выполняет поиск в словарях пространств имен экземпляров и классов, но классы могут также получать линию поведения от своего типа, который не виден поиску при нормальном наследовании.

Чтобы заложить основу для понимания такого отличия, в следующем разделе описана процедура, которую интерпретатор Python придерживается для реализации отношения "экземпляр типа".

Протокол оператора class

Создание подклассов класса type для его настройки – на самом деле лишь половина магии, стоящей за метаклассами. Нам по-прежнему необходимо как-то направлять создание класса метаклассу вместо стандартного type. Для полного понимания, каким образом все организовано, нам также нужно знать, как операторы class делают свою работу.

Мы уже выяснили, что когда интерпретатор Python добирается до оператора class, он выполняет его вложенный блок кода, чтобы создать атрибуты – все имена, которым присваиваются значения на верхнем уровне вложенного блока кода, становятся атрибутами в результирующем объекте класса. Такими именами обычно являются функции методов, создаваемые вложенными операторами def, но они также могут быть произвольными атрибутами, которым присваиваются значения для создания данных класса, разделяемых всеми экземплярами.

Говоря формально, чтобы это произошло, интерпретатор Python следует стандартному протоколу: в *конце оператора* class и после выполнения всего вложенного в него кода в словаре пространств имен, соответствующем локальной области видимости класса, Python обращается к объекту type для создания объекта *класс*:

```
класс = type(имя_класса, суперклассы, словарь_атрибутов)
```

В type определен метод перегрузки операций __call__, который при вызове объекта type по очереди запускает два других метода:

```
type.__new__(класс_type, имя_класса, суперклассы, словарь_атрибутов)
type.__init__(класс, имя_класса, суперклассы, словарь_атрибутов)
```

Метод `__new__` создает и возвращает новый объект *класс*, после чего метод `__init__` инициализирует вновь созданный объект. Как вскоре будет показано, именно они выступают в качестве привязок, которые метаклассы, являющиеся подклассами type, обычно применяют для настройки классов.

Скажем, пусть имеется определение класса Spam следующего вида:

```
class Eggs: ...                       # Здесь находятся наследуемые имена

class Spam(Eggs):                     # Наследуется от Eggs
    data = 1                          # Атрибут данных класса
    def meth(self, arg):              # Атрибут метода класса
        return self.data + arg
```

Интерпретатор Python внутренне запустит вложенный блок кода для создания двух атрибутов класса (data и meth) и затем обратится к объекту type, чтобы сгенерировать объект класса в конце оператора class:

```
Spam = type('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

На самом деле вы можете вызвать type подобным образом самостоятельно и создать класс динамически — хотя здесь с поддельной функцией метода и пустым кортежем суперклассов (object добавляется автоматически в Python 3.X и 2.X):

```
>>> x = type('Spam', (), {'data': 1, 'meth': (lambda x, y: x.data + y)})
>>> i = x()
>>> x, i
(<class '__main__.Spam'>, <__main__.Spam object at 0x029E7780>)
>>> i.data, i.meth(2)
(1, 3)
```

Созданный класс оказывается точно таким же, как полученный в результате выполнения оператора class:

```
>>> x.__bases__
(<class 'object'>,)
>>> [(a, v) for (a, v) in x.__dict__.items() if not a.startswith('__')]
[('data', 1), ('meth', <function <lambda> at 0x0297A158>)]
```

Тем не менее, поскольку вызов type производится автоматически в конце оператора class, он представляет собой идеальную привязку для дополнения или другой обработки класса. Хитрость заключается в замене стандартного класса type специальным подклассом, который будет перехватывать такой вызов. В следующем разделе показано, как это сделать.

Объявление метаклассов

Как вы только что видели, по умолчанию классы создаются посредством класса type. Чтобы сообщить интерпретатору Python о необходимости создания класса с помощью специального метакласса, вам просто понадобится объявить метакласс для перехвата нормального вызова, создающего экземпляр, в определяемом пользователем классе. Способ достижения цели зависит от используемой версии Python.

Объявление в Python 3.X

В Python 3.X желаемый метакласс указывается как *ключевой* аргумент в заголовке оператора class:

```
class Spam(metaclass=Meta):           # Версия Python 3.X (только)
```

Наследуемые суперклассы тоже можно перечислять в заголовке оператора class. Например, определяемый ниже новый класс Spam унаследован от суперкласса Eggs, но также является экземпляром, созданным метаклассом Meta:

```
class Spam(Eggs, metaclass=Meta):      # Допускаются обычные суперклассы:
                                       #   должны указываться первыми
```

В такой форме суперклассы должны указываться перед метаклассом; фактически здесь применяются правила упорядочения, используемые для ключевых аргументов в вызовах функций.

Объявление в Python 2.X

Мы можем достичь того же эффекта и в Python 2.X, но указывая метакласс по-другому — вместо ключевого аргумента применяя *атрибут класса*:

```
class Spam(object):           # Версия Python 2.X (только), object необязателен?
    __metaclass__ = Meta

class Spam(Eggs, object):     # Допускаются обычные суперклассы:
                              #   подразумевается object
    __metaclass__ = Meta
```

Формально некоторые классы в Python 2.X вовсе *не* обязаны явно наследоваться от object, чтобы задействовать метаклассы. Обобщенный механизм координирования метаклассов появился одновременно с классами нового стиля, но сам он к ним не привязан. Однако он их *производит* — при наличии объявления __metaclass__ интерпретатор Python 2.X автоматически делает результирующий класс классом нового стиля, добавляя object в его последовательность __bases__. При отсутствии такого объявления интерпретатор Python 2.X просто использует инструмент создания классических классов в качестве стандартного метакласса. По этой причине некоторые классы в Python 2.X требуют только атрибута __metaclass__.

С другой стороны, метаклассы *подразумевают*, что ваш класс будет классом нового стиля в Python 2.X, даже без явного наследования от object. Они будут вести себя несколько иначе в сравнении с тем, как было обрисовано в главе 32. К тому же вам предстоит увидеть, что Python 2.X может требовать, чтобы они либо их суперклассы явно наследовались от object, поскольку в подобном контексте классу нового стиля нельзя иметь только классические суперклассы. С учетом этого наследование от object не помешает как своего рода предупреждение о природе класса и во избежание потенциальных проблем может считаться обязательным.

Также в Python 2.X доступна глобальная переменная __metaclass__ уровня *модуля* для связывания всех классов в модуле с метаклассом. В Python 3.X она больше не поддерживается, т.к. задумывалась в качестве временной меры, чтобы облегчить переход к применению по умолчанию классов нового стиля, не наследуя каждый класс от object. Кроме того, в Python 3.X также игнорируется атрибут класса Python 2.X, а форма с ключевым аргументом из Python 3.X трактуется как синтаксическая ошибка в Python 2.X, и потому простой путь к обеспечению переносимости отсутствует. Тем не менее, за исключением отличающегося синтаксиса объявление метакласса в Python 2.X и 3.X дает один и тот же результат, который мы обсудим далее.

Координирование метаклассов в Python 3.X и 2.X

Когда конкретный метакласс объявлен в соответствии с синтаксисом, описанным в предшествующих разделах, вызов для создания объекта *класс*, выполняемый в конце оператора `class`, модифицируется так, чтобы обращаться к метаклассу, а не к `type`:

```
класс = Meta(имя_класса, суперклассы, словарь_атрибутов)
```

Из-за того, что метакласс является подклассом `type`, метод `__call__` класса `type` делегирует вызовы для создания и инициализации нового объекта *класс* метаклассу, если в нем определены специальные версии следующих методов:

```
Meta.__new__(Meta, имя_класса, суперклассы, словарь_атрибутов)
Meta.__init__(класс, имя_класса, суперклассы, словарь_атрибутов)
```

В целях демонстрации ниже приведен пример из предыдущего разделах, дополненный спецификацией метаклассов Python 3.X:

```
class Spam(Eggs, metaclass=Meta):        # Наследуется от Eggs, экземпляр Meta
    data = 1                             # Атрибут данных класса
    def meth(self, arg):                 # Атрибут метода класса
        return self.data + arg
```

В конце этого оператора `class` интерпретатор Python внутренне запускает следующий код, чтобы создать объект класса — опять-таки вызов, который вы могли бы сделать вручную, но он автоматически выполняется механизмом оператора `class`:

```
Spam = Meta('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

Если метакласс определяет собственные версии `__new__` и `__init__`, тогда они будут вызываться по очереди унаследованным из класса `type` методом `__call__`, чтобы создать и инициализировать новый класс. Совокупный эффект заключается в том, что происходит автоматический запуск методов, предоставляемых метаклассом, как часть процесса создания класса. В следующем разделе показано, каким образом мы можем приступить к решению финальной части головоломки, связанной с метаклассами.

В настоящей главе для метаклассов используется синтаксис с ключевыми аргументами Python 3.X, но не синтаксис с атрибутом класса Python 2.X. Читателям, работающим с Python 2.X, придется соответствующим образом переводить код. Дело в том, что обеспечить нейтральность к версиям здесь непросто — Python 3.X не распознает атрибут класса, а Python 2.X не допускает синтаксис с ключевыми аргументами — и указание для каждого примера двух листингов не решит проблему переносимости (зато увеличит размер главы!).

Реализация метаклассов

До сих пор мы видели, что Python направляет вызовы для создания классов метаклассу, если он указан и подготовлен. Однако как на самом деле мы будем реализовывать метакласс, который настраивает `type`?

Оказывается, что большая часть истории вам уже известна — метаклассы реализуются с помощью нормальных операторов `class` и семантики Python. По определению это просто классы, унаследованные от `type`. Существенные отличия заключаются лишь в том, что Python вызывает их *автоматически* в конце оператора `class`, и что они обязаны придерживаться *интерфейса*, ожидаемого суперклассом `type`.

Базовый метакласс

Возможно, самым простым метаклассом будет подкласс `type` с методом `__new__`, который создает объект класса, запуская стандартную версию в `type`. Метод `__new__` такого метакласса запускается методом `__call__`, унаследованным из `type`; он обычно выполняет любую требующуюся настройку и вызывает метод `__new__` суперкласса `type`, чтобы создать и возвратить новый объект класса:

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        # Запускается унаследованным методом type.__call__
        return type.__new__(meta, classname, supers, classdict)
```

В действительности этот метакласс ничего не делает (мы могли бы также позволить создать класс стандартному классу `type`), но он демонстрирует способ перехвата привязки метакласса с целью настройки. Поскольку метакласс вызывается в конце оператора `class`, а метод `__call__` объекта `type` координируется для вызова методов `__new__` и `__init__`, предоставленный нами код в методах способен управлять всеми классами, создаваемыми из метакласса.

Ниже снова приводится наш пример, где в метакласс и класс добавлены операторы вывода, предназначенные для отслеживания:

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
      print('In MetaOne.new:', meta, classname, supers, classdict, sep='\n...')
      return type.__new__(meta, classname, supers, classdict)

class Eggs:
    pass

print('making class')                       # создание класса
class Spam(Eggs, metaclass=MetaOne):        # Наследуется от Eggs, экземпляр MetaOne
    data = 1                                # Атрибут данных класса
    def meth(self, arg):                    # Атрибут метода класса
        return self.data + arg

print('making instance')                    # создание экземпляра
X = Spam()
print('data:', X.data, X.meth(2))
```

Здесь класс `Spam` наследуется от `Eggs` и является экземпляром `MetaOne`, но `X` — экземпляр `Spam`. При запуске кода в Python 3.X обратите внимание, что метакласс вызывается в конце оператора `class` перед созданием экземпляра — метаклассы предназначены для обработки *классов*, а классы ориентированы на обработку нормальных *экземпляров*:

```
c:\code> py -3 metaclass1.py
making class
In MetaOne.new:
...<class '__main__.MetaOne'>
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x02A191E0>, '__module__':
'__main__'}
making instance
data: 1 3
```

Замечание по представлению. Ради экономии места в этой главе адреса приводятся в сокращенном виде, а некоторые несущественные встроенные имена __X__ в словарях пространств имен опускаются. Кроме того, как отмечалось выше, из-за отличающегося синтаксиса объявления здесь не преследуется цель обеспечить переносимость в Python 2.X. Для запуска в Python 2.X применяйте форму с атрибутом класса и при желании измените операции вывода. Пример работает в Python 2.X с показанными ниже модификациями (файл metaclass1-2x.py). Обратите внимание на то, что класс Eggs *или* Spam должен быть явно унаследован от object, иначе интерпретатор Python 2.X выдаст предупреждение, т.к. класс нового стиля не может иметь только классические базовые классы – при наличии сомнений используйте object в клиентах метаклассов Python 2.X:

```
from __future__ import print_function    # Для Python 2.X (только)
class Eggs(object):                     # Одно из указаний object необязательно
class Spam(Eggs, object):
   __metaclass__ = MetaOne
```

Настройка создания и инициализации

Метаклассы также способны подключаться к протоколу __init__, запускаемому методом __call__ объекта type. В общем случае метод __new__ создает и возвращает объект класса, а метод __init__ инициализирует уже созданный класс, передаваемый в качестве аргумента. Метаклассы могут применять любую из двух или обе привязки для управления классом на стадии создания:

```
class MetaTwo(type):
   def __new__(meta, classname, supers, classdict):
      print('In MetaTwo.new: ', classname, supers, classdict, sep='\n...')
      return type.__new__(meta, classname, supers, classdict)

   def __init__(Class, classname, supers, classdict):
      print('In MetaTwo.init:', classname, supers, classdict, sep='\n...')
      print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
   pass

print('making class')
class Spam(Eggs, metaclass=MetaTwo):    # Наследуется от Eggs, экземпляр MetaTwo
   data = 1                             # Атрибут данных класса
   def meth(self, arg):                 # Атрибут метода класса
      return self.data + arg

print('making instance')
X = Spam()
print('data:', X.data, X.meth(2))
```

В данной ситуации метод инициализации класса запускается после метода создания класса, но оба метода вызываются в конце оператора class до создания любых экземпляров. И наоборот, метод __init__ в Spam будет выполняться во время создания *экземпляра*, а метод __init__ метакласса не затрагивает и не запускает его:

```
c:\code> py -3 metaclass2.py
making class
In MetaTwo.new:
...Spam
...(<class '__main__.Eggs'>,)
```

```
...{'data': 1, 'meth': <function Spam.meth at 0x02967268>, '__module__':
'__main__'}
In MetaTwo.init:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x02967268>, '__module__':
'__main__'}
...init class object: ['__qualname__', 'data', '__module__', 'meth', '__doc__']
making instance
data: 1 3
```

Другие методики реализации метаклассов

Хотя переопределение методов `__new__` и `__init__` суперкласса `type` являются самым распространенным способом вставить логику внутрь процесса создания объектов классов с привязкой к метаклассу, возможны и другие схемы.

Использование простых фабричных функций

Например, в действительности метаклассы вообще не обязаны быть классами. Как уже известно, оператор `class` выдает простой вызов для создания класса в заключение его обработки. Из-за этого в качестве метакласса в принципе может применяться *любой вызываемый объект* при условии, что он принимает переданные аргументы и возвращает объект, совместимый с целевым классом. Фактически простая фабричная функция может справиться с задачей наравне с подклассом `type`:

```
# Простая функция тоже может служить в качестве метакласса

def MetaFunc(classname, supers, classdict):
    print('In MetaFunc: ', classname, supers, classdict, sep='\n...')
    return type(classname, supers, classdict)

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaFunc):  # В конце запускается простая функция
    data = 1                           # Функция возвращает класс
    def meth(self, arg):
        return self.data + arg

print('making instance')
X = Spam()
print('data:', X.data, X.meth(2))
```

Функция `MetaFunc` вызывается в конце оператора `class` и возвращает ожидаемый новый объект класса. Она просто перехватывает вызов, который по умолчанию перехватывается методом `__call__` объекта `type`:

```
c:\code> py -3 metaclass3.py
making class
In MetaFunc:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x029471E0>, '__module__':
'__main__'}
making instance
data: 1 3
```

Перегрузка операций вызова, создающих классы, с помощью нормальных классов

Поскольку экземпляры нормальных классов способны реагировать на операции вызова посредством перегрузки операций, они также могут исполнять некоторые роли метаклассов во многом подобно функции из предыдущего раздела. Вывод приведенной далее версии аналогичен выводу предшествующих версий, но она основана на простом классе, который вообще не наследуется от `type` и предоставляет своим экземплярам метод `__call__`, перехватывающий обращения к метаклассу с использованием обычной перегрузки операций. Обратите внимание, что методы `__new__` и `__init__` должны здесь иметь отличающиеся имена, иначе они будут запускаться при *создании* экземпляра `Meta`, а не когда позже он вызывается в роли метакласса:

```
# Экземпляр нормального класса тоже может служить метаклассом

class MetaObj:
    def __call__(self, classname, supers, classdict):
        print('In MetaObj.call: ', classname, supers, classdict, sep='\n...')
        Class = self.__New__(classname, supers, classdict)
        self.__Init__(Class, classname, supers, classdict)
        return Class

    def __New__(self, classname, supers, classdict):
        print('In MetaObj.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classdict):
        print('In MetaObj.init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaObj()):  # MetaObj - экземпляр нормального класса
    data = 1                            # Вызывается в конце оператора class
    def meth(self, arg):
        return self.data + arg

print('making instance')
X = Spam()
print('data:', X.data, X.meth(2))
```

Во время выполнения вызовы трех методов координируются через метод `__call__` экземпляра, унаследованный из нормального класса, но без какой-либо зависимости от механизма или семантики `type`:

```
c:\code> py -3 metaclass4.py
making class
In MetaObj.call:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x029492F0>, '__module__':
'__main__'}
In MetaObj.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x029492F0>, '__module__':
'__main__'}
```

```
In MetaObj.init:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x029492F0>, '__module__': '__main__'}
...init class object: ['__module__', '__doc__', 'data', '__qualname__', 'meth']
making instance
data: 1 3
```

На самом деле в этой кодовой модели мы можем применять обычное наследование от суперклассов для получения метода перехвата вызовов — суперкласс здесь исполняет в точности ту же самую роль, что и `type`, во всяком случае, с точки зрения координирования метаклассов:

```
# Экземпляры нормально наследуют метод перехвата вызовов из классов
#   и их суперклассов
class SuperMetaObj:
    def __call__(self, classname, supers, classdict):
        print('In SuperMetaObj.call: ', classname, supers, classdict, sep='\n...')
        Class = self.__New__(classname, supers, classdict)
        self.__Init__(Class, classname, supers, classdict)
        return Class

class SubMetaObj(SuperMetaObj):
    def __New__(self, classname, supers, classdict):
        print('In SubMetaObj.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classdict):
        print('In SubMetaObj.init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Spam(Eggs, metaclass=SubMetaObj()):   # Обращается к экземпляру Sub
                                            #   через Super.__call__
    ...остальной код не изменился...

c:\code> py -3 metaclass4-super.py
making class
In SuperMetaObj.call:
...как и ранее...
In SubMetaObj.new:
...как и ранее...
In SubMetaObj.init:
...как и ранее...
making instance
data: 1 3
```

Несмотря на то что показанные альтернативные формы работоспособны, большинство метаклассов выполняют свою работу, переопределяя методы `__new__` и `__init__` суперкласса `type`; на практике такого объема контроля вполне достаточно, и результирующий код зачастую получается проще, чем в других схемах. Кроме того, метаклассы имеют доступ к дополнительным инструментам, таким как *методы* классов, которые мы рассмотрим позже, и это может оказывать более прямое влияние на поведение классов, нежели ряд других схем.

Тем не менее, далее вы увидите, что простой метакласс, основанный на вызываемом объекте, часто способен работать во многом подобно декоратору классов, что позволяет метаклассам управлять не только классами, но и экземплярами. Однако сначала в следующем разделе будет представлен пример, демонстрирующий концепции распознавания имен метаклассами.

Перегрузка операций вызова, создающих классы, с помощью метаклассов

Поскольку метаклассы принимают участие в нормальных механизмах ООП, они также способны напрямую перехватывать операцию вызова, создающую класс в конце оператора `class`, за счет переопределения метода `__call__` объекта `type`. При переопределении методов `__new__` и `__call__` важно не забывать о вызове их стандартных версий в `type`, если они предназначены для создания класса в конце, а метод `__call__` должен обращаться к `type`, чтобы запустить другие два метода:

```
# Классы тоже могут перехватывать вызовы (но встроенные
# операции ищутся в метаклассах, а не в суперклассах!)

class SuperMeta(type):
   def __call__(meta, classname, supers, classdict):
      print('In SuperMeta.call: ', classname, supers, classdict, sep='\n...')
      return type.__call__(meta, classname, supers, classdict)

   def __init__(Class, classname, supers, classdict):
      print('In SuperMeta init:', classname, supers, classdict, sep='\n...')
      print('...init class object:', list(Class.__dict__.keys()))

print('making metaclass')
class SubMeta(type, metaclass=SuperMeta):
   def __new__(meta, classname, supers, classdict):
      print('In SubMeta.new: ', classname, supers, classdict, sep='\n...')
      return type.__new__(meta, classname, supers, classdict)

   def __init__(Class, classname, supers, classdict):
      print('In SubMeta init:', classname, supers, classdict, sep='\n...')
      print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
   pass

print('making class')
class Spam(Eggs, metaclass=SubMeta):     # Обращается к SubMeta
                                         #   через SuperMeta.__call__
   data = 1
   def meth(self, arg):
      return self.data + arg

print('making instance')
X = Spam()
print('data:', X.data, X.meth(2))
```

В коде присутствует несколько странностей, которые вскоре будут объяснены. Тем не менее, во время выполнения все три переопределенных метода по очереди запускаются для `Spam`, как было в предыдущем разделе. По существу это снова то, что по умолчанию делает объект `type`, но имеется дополнительный вызов метакласса для подкласса метакласса (*метаподкласса?*):

```
c:\code> py -3 metaclass5.py
making metaclass
In SuperMeta init:
...SubMeta
...(<class 'type'>,)
...{'__init__': <function SubMeta.__init__ at 0x028F92F0>, ...}
...init class object: ['__doc__', '__module__', '__new__', '__init__, ...]
making class
In SuperMeta.call:
```

```
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x028F9378>, '__module__':
'__main__'}
In SubMeta.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x028F9378>, '__module__':
'__main__'}
In SubMeta init:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x028F9378>, '__module__':
'__main__'}
...init class object: ['__qualname__', '__module__', '__doc__', 'data', 'meth']
making instance
data: 1 3
```

Пример усложняется тем фактом, что в нем переопределяется метод, вызываемый *встроенной* операцией — в данном случае вызов запускается автоматически для создания класса. Метаклассы используются для создания объектов классов, но при вызове в роли метаклассов лишь генерируют экземпляры самих себя. По указанной причине поиск имен при наличии метаклассов может несколько отличаться о того, к чему мы привыкли. Скажем, метод `__call__` ищется встроенными операциями в классе (т.е. типе) объекта; для метаклассов это означает метакласс метакласса!

Далее будет показано, что метаклассы также нормально *наследуют* имена из других метаклассов, но, как и в случае обычных классов, похоже, это применимо только к *явным* извлечениям имен, а не к *неявному* поиску имен для встроенных операций наподобие вызовов. Последнее выглядит как просмотр *класса* метакласса, доступного в его ссылке `__class__`, которой будет либо стандартный `type`, либо метакласс. Здесь возникает та же самая проблема координирования встроенных операций, которую мы часто встречали в книге при работе с экземплярами нормальных классов. Для установки такой ссылки требуется ключевой аргумент `metaclass` в `SubMeta`, хотя он также инициирует шаг создания метакласса для самого метакласса.

Проследите все вызовы в выводе. Метод `__call__` из `SuperMeta` *не* запускается для вызова `SuperMeta` при создании `SubMeta` (взамен это направляется `type`), но *запускается* для вызова `SubMeta` при создании `Spam`. Обычного наследования от `SuperMeta` не будет достаточно для перехвата вызовов `SubMeta`, и по причинам, которые мы увидим позже, поступать так с методами перегрузки операций на самом деле неправильно: затем `Spam` получает метод `__call__` из `SuperMeta`, приводя к тому, что вызовы для создания экземпляров `Spam` потерпят неудачу до того, как будет создан хоть какой-нибудь экземпляр. Тонко, но верно!

Вот иллюстрация проблемы в более простых терминах — нормальный суперкласс пропускается для *встроенных имен*, но не для *явных* извлечений и вызовов; последние полагаются на обычное наследование имен атрибутов:

```
class SuperMeta(type):
    def __call__(meta, classname, supers, classdict):     # По имени,
                                                          #   не встроенное
        print('In SuperMeta.call:', classname)
        return type.__call__(meta, classname, supers, classdict)
```

```
class SubMeta(SuperMeta):                         # Создается стандартным type
   def __init__(Class, classname, supers, classdict):     # Переопределение
                                                          #   type.__init__
      print('In SubMeta init:', classname)

print(SubMeta.__class__)
print([n.__name__ for n in SubMeta.__mro__])
print()
print(SubMeta.__call__)       # Не дескриптор данных, если найден по имени
print()
SubMeta.__call__(SubMeta, 'xxx', (), {})    # Явные вызовы работают:
                                            #   наследование классов
print()
SubMeta('yyy', (), {})        # Но неявные обращения к встроенным именам
                              #   не работают: type

c:\code> py -3 metaclass5b.py
<class 'type'>
['SubMeta', 'SuperMeta', 'type', 'object']

<function SuperMeta.__call__ at 0x029B9158>

In SuperMeta.call: xxx
In SubMeta init: xxx

In SubMeta init: yyy
```

Разумеется, рассмотренный конкретный пример является особым случаем: перехват встроенной операции, выполняемой на метаклассе, вероятно, будет тем редким сценарием использования, связанным с `__call__`. Но это подчеркивает основную асимметрию и явную противоречивость: *нормальное наследование атрибутов не задействуется в полной мере при координировании встроенных операций* — как для экземпляров, так и для классов.

Однако чтобы по-настоящему понять тонкости приведенного выше примера, необходимо получить более формальное представление о том, что метаклассы означают для распознавания имен Python в целом.

Наследование и экземпляр

Поскольку метаклассы указываются способами, похожими на указание наследуемых суперклассов, поначалу они могут слегка сбивать с толку. Описанные ниже ключевые моменты помогут подытожить и прояснить модель.

Метаклассы наследуются от класса* `type` *(обычно)

Несмотря на то что метаклассы исполняют специальную роль, они реализуются посредством операторов `class` и следуют обычной модели ООП в Python. Например, будучи подклассами `type`, они могут переопределять методы объекта `type`, настраивая их должным образом. Метаклассы, как правило, переопределяют методы `__new__` и `__init__` класса `type` для настройки создания и инициализации классов. Хотя и реже, они могут переопределять также метод `__call__`, если требуется напрямую перехватывать вызов создания класса в конце (пусть и со сложностями, изложенными в предыдущем разделе). Метаклассы могут даже быть простыми функциями или другими вызываемыми объектами, возвращающими произвольные объекты, а не подклассами `type`.

Объявления метаклассов наследуется подклассами

Объявление `metaclass=метакласс` в определяемом пользователем классе *наследуется* его нормальными подклассами, так что метакласс будет запускаться для создания каждого класса, который наследует эту спецификацию в цепочке наследования суперклассов.

Атрибуты метаклассов не наследуются экземплярами классов

Объявления метаклассов указывают отношение между *экземплярами*, которое отличается от того, что мы называли наследованием до сих пор. Поскольку классы являются экземплярами метаклассов, определяемое метаклассом поведение применяется к классу, но не к создаваемым впоследствии экземплярам класса. Экземпляры получают поведение от своих классов и суперклассов, но не от метаклассов. Формально процедура наследования атрибутов для обычных экземпляров выполняет поиск только в словарях `__dict__` экземпляра, его класса и всех суперклассов класса; для обычных экземпляров метаклассы в поиск при наследовании *не* включаются.

Атрибуты метаклассов получаются классами

Напротив, классы *получают* методы своих метаклассов благодаря отношению между экземплярами. Это источник поведения классов, которое обрабатывает сами классы. Формально классы обзаводятся атрибутами метаклассов посредством своих ссылок `__class__` в точности, как нормальные экземпляры получают имена от своих классов, но сначала предпринимается попытка наследования через поиск в `__dict__`: когда одно и то же имя доступно классу в метаклассе *и* в суперклассе, то используется версия из суперкласса (через наследование), а не из метакласса (через экземпляр). Тем не менее, атрибут `__class__` класса не следует в его собственные экземпляры: атрибуты метаклассов делаются доступными их классам-экземплярам, но не экземплярам этих классов-экземпляров (тут снова уместно сослаться на Доктора Сьюза...).

Возможно, перечисленные выше моменты будет легче понять, написав код. В целях демонстрации рассмотрим такой пример:

```
# Файл metainstance.py

class MetaOne(type):
    def __new__(meta, classname, supers, classdict):  # Переопределение
                                                       #   метода type
        print('In MetaOne.new:', classname)
        return type.__new__(meta, classname, supers, classdict)
    def toast(self):
        return 'toast'

class Super(metaclass=MetaOne): # Метакласс наследуется также и подклассами
    def spam(self):    # MetaOne запускается дважды для двух классов
        return 'spam'

class Sub(Super):      # Суперкласс: наследование или отношение между экземплярами
    def eggs(self):    # Классы наследуют атрибуты от суперклассов
        return 'eggs'  # Но не от метаклассов
```

Когда такой код запускается (как сценарий или модуль), метакласс обрабатывает создание для *обоих* клиентских классов, а *экземпляры* наследуют атрибуты класса, но *не* атрибуты метакласса:

```
>>> from metainstance import *      # Выполняются операторы class:
                                    #   метакласс запускается дважды
In MetaOne.new: Super
In MetaOne.new: Sub
>>> X = Sub()       # Нормальный экземпляр класса, определяемого пользователем
>>> X.eggs()        # Унаследован из Sub
'eggs'
>>> X.spam()        # Унаследован из Super
'spam'
>>> X.toast()       # Не наследуется из метакласса
AttributeError: 'Sub' object has no attribute 'toast'
Ошибка атрибута: объект Sub не имеет атрибута toast
```

В противоположность этому *классы* наследуют имена от своих суперклассов и обзаводятся именами из метакласса (который в данном примере *сам* унаследован из суперкласса):

```
>>> Sub.eggs(X)       # Собственный метод
'eggs'
>>> Sub.spam(X)       # Унаследован из Super
'spam'
>>> Sub.toast()       # Получен из метакласса
'toast'
>>> Sub.toast(X)      # Не метод нормального класса
TypeError: toast() takes 1 positional argument but 2 were given
Ошибка типа: toast() принимает 1 позиционный аргумент, но было передано 2
```

Обратите внимание, что последний из предшествующих вызовов терпит неудачу, когда мы передаем экземпляр, поскольку имя распознается как метод метакласса, а не метод нормального класса. Фактически и объект, из которого извлекается имя, и его источник становятся здесь решающими. Методы, полученные из метакласса, привязываются к целевому *классу*, в то время как методы из нормальных классов являются *несвязанными*, если извлекаются через класс, но *связанными*, когда извлекаются через экземпляр:

```
>>> Sub.toast
<bound method MetaOne.toast of <class 'metainstance.Sub'>>
>>> Sub.spam
<function Super.spam at 0x0298A2F0>
>>> X.spam
<bound method Sub.spam of <metainstance.Sub object at 0x02987438>>
```

Последние два правила изучались ранее в главе 31 при рассмотрении связанных методов; первое правило новое, но напоминает одно из правил для методов класса. Чтобы понять, почему все работает именно так, нам необходимо заняться исследованием также отношения между экземплярами.

Метакласс или суперкласс

Давайте обратимся к более доступной форме. Взгляните, что происходит в показанном ниже взаимодействии: будучи *экземпляром* метакласса A, класс B получает атрибут из A, но этот атрибут не делается доступным для наследования собственными экземплярами класса B — получение имен экземплярами метакласса отличается от нормального наследования, применяемого для экземпляров класса:

```
>>> class A(type): attr = 1
>>> class B(metaclass=A): pass      # B - экземпляр метакласса и получает
                                    #   атрибут attr из метакласса
>>> I = B()           # I наследует атрибуты из класса, но не из метакласса!
>>> B.attr
1
>>> I.attr
AttributeError: 'B' object has no attribute 'attr'
Ошибка атрибута: объект B не имеет атрибута attr
>>> 'attr' in B.__dict__, 'attr' in A.__dict__
(False, True)
```

По контрасту с этим, если трансформировать A из метакласса в суперкласс, тогда имена, *унаследованные* из суперкласса A, становятся доступными для создаваемых позже экземпляров класса B и обнаруживаются путем поиска внутри словарей пространств имен в классах в дереве — т.е. за счет проверки `__dict__` объектов в порядке распознавания методов (MRO), что очень похоже на пример `mapattrs.py` из главы 32:

```
>>> class A: attr = 1
>>> class B(A): pass      # I наследует атрибуты из класса и суперклассов
>>> I = B()
>>> B.attr
1
>>> I.attr
1
>>> 'attr' in B.__dict__, 'attr' in A.__dict__
(False, True)
```

Вот почему метаклассы часто выполняют свою работу, манипулируя словарем пространств имен нового класса, если им нужно повлиять на поведение последующих объектов экземпляров — экземпляры будут видеть имена в классе, но не в его метаклассе. Однако посмотрите, что происходит, если идентичное имя доступно в *обоих* источниках атрибутов — вместо имени, полученного из экземпляра, используется *унаследованное* имя:

```
>>> class M(type): attr = 1
>>> class A: attr = 2
>>> class B(A, metaclass=M): pass  # Суперклассы имеют приоритет
                                   #   над метаклассами
>>> I = B()
>>> B.attr, I.attr
(2, 2)
>>> 'attr' in B.__dict__, 'attr' in A.__dict__, 'attr' in M.__dict__
(False, True, True)
```

Это верно независимо от относительной высоты источников наследования и экземпляров — интерпретатор Python проверяет `__dict__` каждого класса в порядке MRO (*наследование*), прежде чем переходить к получению из экземпляра (*отношение между экземплярами*):

```
>>> class M(type): attr = 1
>>> class A: attr = 2
>>> class B(A): pass
>>> class C(B, metaclass=M): pass  # Суперкласс на два уровня выше метакласса:
                                   #   все равно выигрывает
>>> I = C()
```

```
>>> I.attr, C.attr
(2, 2)
>>> [x.__name__ for x in C.__mro__]    # Сведения MRO ищите в главе 32
['C', 'B', 'A', 'object']
```

В действительности классы получают атрибуты метаклассов через свои ссылки `__class__` тем же самым способом, каким нормальные экземпляры наследуют их из классов через свои атрибуты `__class__`, что имеет смысл, поскольку классы также являются экземплярами метаклассов. Главное отличие заключается в том, что наследование экземпляра не проходит по ссылке `__class__` класса, но взамен ограничивает свой охват словарем `__dict__` каждого класса в дереве согласно MRO – следуя только `__bases__` на уровне каждого класса и применяя ссылку `__class__` экземпляра только раз:

```
>>> I.__class__              # Следует наследованию: класс экземпляра
<class '__main__.C'>
>>> C.__bases__              # Следует наследованию: суперклассы класса
(<class '__main__.B'>,)
>>> C.__class__              # Следует получению из экземпляра: метакласс
<class '__main__.M'>
>>> C.__class__.attr         # Еще один способ добраться до атрибутов метакласса
1
```

После изучения всего того, что приводилось выше, возможно вы заметите почти явную симметрию, которая подводит нас к теме следующего раздела.

Наследование: вся история

Как выясняется, наследование экземпляра работает аналогично независимо от того, создан "экземпляр" из нормального класса или представляет собой класс, созданный из метакласса, являющегося подклассом `type`. Такое единственное правило поиска атрибутов благоприятствует более широкому и похожему понятию иерархий наследования метаклассов. Для иллюстрации основ этого концептуального объединения в приведенном ниже взаимодействии экземпляр наследует атрибуты из всех своих классов, класс – из классов и метаклассов, а метаклассы – из более высоких метаклассов (*суперметаклассов?*):

```
>>> class M1(type): attr1 = 1                # Дерево наследования метаклассов
>>> class M2(M1): attr2 = 2                  # Получает имена __bases__, __
class__, __mro__

>>> class C1: attr3 = 3                      # Дерево наследования суперклассов
>>> class C2(C1,metaclass=M2): attr4 = 4     # Получает имена __bases__,
                                             #   __class__, __mro__

>>> I = C2()                        # I получает __class__, но не остальные имена
>>> I.attr3, I.attr4                # Экземпляр наследует имена из дерева суперклассов
(3, 4)
>>> C2.attr1, C2.attr2, C2.attr3, C2.attr4    # Класс получает имена
                                              #   из обоих деревьев!
(1, 2, 3, 4)
>>> M2.attr1, M2.attr2                        # Метакласс тоже наследует имена!
(1, 2)
```

Оба пути наследования – из класса и из метакласса – задействуют те же самые ссылки, хотя не рекурсивно; экземпляры не наследуют имена метакласса своего класса, но могут запросить их явно:

```
>>> I.__class__                # Ссылки следуют на экземпляр без __bases__
<class '__main__.C2'>
>>> C2.__bases__
(<class '__main__.C1'>,)

>>> C2.__class__               # Ссылки следуют на класс после __bases__
<class '__main__.M2'>
>>> M2.__bases__
(<class '__main__.M1'>,)

>>> I.__class__.attr1          # Направление наследования на дерево
                               #  метаклассов класса
1
>>> I.attr1                    # Хотя __class__ класса нормально не проходится
AttributeError: 'C2' object has no attribute 'attr1'
Ошибка атрибута: объект C2 не имеет атрибута attr1

>>> M2.__class__               # Оба дерева имеют MRO и ссылки на экземпляры
<class 'type'>
>>> [x.__name__ for x in C2.__mro__]    # Дерево __bases__ из I.__class__
['C2', 'C1', 'object']
>>> [x.__name__ for x in M2.__mro__]    # Дерево __bases__ из C2.__class__
['M2', 'M1', 'type', 'object']
```

Если вас интересуют метаклассы или вы должны использовать код с ними, тогда изучите предложенные примеры еще раз. В сущности, наследование следует по `__bases__` перед переходом к единственному `__class__`, нормальные экземпляры не имеют `__bases__`, а классы имеют то и другое – нормальные они или метаклассы. На самом деле этот пример важно понять, чтобы освоить распознавание имен Python в целом, как объясняется в следующем разделе.

Алгоритм наследования Python: простая версия

Теперь, когда вам известно о метаклассах, мы в состоянии окончательно формализовать правила наследования, которые они дополняют. Формально наследование вводит в действие две разных, но похожих процедуры поиска, и основано на MRO. Поскольку `__bases__` применяется для создания упорядочения `__mro__` во время создания классов, а `__mro__` класса включает самого себя, обобщение из предыдущего раздела будет таким же, как приведенное далее начальное определение алгоритма наследования нового стиля Python.

Для поиска явного имени атрибута выполнить описанные ниже шаги.

1. Начиная с *экземпляра* I, провести поиск в экземпляре, затем в его классе и далее во всех суперклассах класса, используя:
   а) словарь `__dict__` экземпляра I;
   б) словари `__dict__` всех классов в `__mro__`, найденном в `__class__` экземпляра I, слева направо.
2. Начиная с *класса* C, провести поиск в классе, затем во всех его суперклассах и далее в его дереве метаклассов, используя:
   а) словари `__dict__` всех классов в `__mro__`, найденном в самом классе C, слева направо;
   б) словари `__dict__` всех метаклассов в `__mro__`, найденном в `__class__` класса C, слева направо.

3. В шагах 1 и 2 предоставить приоритет *дескрипторам данных*, которые найдены в источниках в пункте б) (см. далее).
4. Для *встроенных* имен в шагах 1 и 2 пропустить пункт а) и начать поиск с пункта б) (см. далее).

Первые два шага выполняются только для обычного явного извлечения атрибутов. Предусмотрены исключения для *встроенных имен* и *дескрипторов*, которые вскоре будут прояснены. Вдобавок, как объяснялось в главе 38, для отсутствующих или всех имен может также применяться метод `__getattr__` или `__getattribute__`.

Большинству программистов нужно знать лишь первое из этих правил и возможно первый шаг второго, которые вместе соответствуют наследованию *классических классов* Python 2.X. Для метаклассов добавлен дополнительный шаг (*2б*), но по существу он такой же, как остальные — безусловно, довольно тонкая равнозначность, но метаклассы не настолько новы, как может показаться. Фактически они — всего лишь один компонент более крупной модели.

Особый случай для дескрипторов

По крайней мере, это нормальный — и *упрощенный* — случай. В предыдущем разделе я специально выделил шаг *3*, т.к. он не применяется к большей части кода и значительно усложняет алгоритм. Тем не менее, оказывается, что в наследовании также предусмотрен особый случай взаимодействия с дескрипторами атрибутов, описанными в главе 38. Если кратко, то дескрипторы, известные как *дескрипторы данных* (те, которые определяют методы `__set__` для перехвата операций присваивания) имеют приоритет, так что их имена переопределяют другие источники наследования.

Такое исключение служит нескольким практическим целям. Например, оно используется для гарантии того, что специальные атрибуты `__class__` и `__dict__` не могут быть переопределены теми же самыми именами в собственном словаре `__dict__` экземпляра:

```
>>> class C: pass                    # Особый случай наследования #1...
>>> I = C()                          # Дескрипторы данных класса имеют приоритет
>>> I.__class__, I.__dict__
(<class '__main__.C'>, {})

>>> I.__dict__['name'] = 'bob'           # Динамические данные в экземпляре
>>> I.__dict__['__class__'] = 'spam'     # Присваивание ключам, не атрибутам
>>> I.__dict__['__dict__'] = {}

>>> I.name                           # I.name поступает из I.__dict__, как обычно
'bob'                                # Но I.__class__ и I.__dict__ - нет!
>>> I.__class__, I.__dict__
(<class '__main__.C'>, {'__class__': 'spam', '__dict__': {}, 'name': 'bob'})
```

Исключительная ситуация с дескрипторами данных проверяется в качестве предварительного шага перед предшествующими двумя правилами наследования. Она более важна для разработчиков интерпретатора Python, чем для программистов на Python, и так или иначе может быть проигнорирована в большинстве прикладного кода — если только *вы сами* не реализуете собственные дескрипторы данных, которые следуют тому же правилу приоритета для особого случая наследования:

```
>>> class D:
        def __get__(self, instance, owner): print('__get__')
        def __set__(self, instance, value): print('__set__')
```

```
>>> class C: d = D()              # Атрибут дескриптора данных
>>> I = C()
>>> I.d                           # Доступ к унаследованному дескриптору данных
__get__
>>> I.d = 1
__set__
>>> I.__dict__['d'] = 'spam'  # Определение того же имени в словаре
                              #   пространств имен экземпляра
>>> I.d                       # Но оно не скрывает дескриптор данных в классе!
__get__
```

И наоборот, если этот дескриптор *не* определяет `__set__`, то имя в словаре экземпляра скроет имя в классе согласно нормальному наследованию:

```
>>> class D:
        def __get__(self, instance, owner): print('__get__')

>>> class C: d = D()
>>> I = C()
>>> I.d                       # Доступ к унаследованному дескриптору не данных
__get__
>>> I.__dict__['d'] = 'spam'  # Скрывает имена в классе согласно правилам
                              #   нормального наследования
>>> I.d
'spam'
```

В обоих случаях интерпретатор Python автоматически запускает метод `__get__` дескриптора, когда он находится по наследованию, а не возвращает сам объект дескриптора — часть магии, относящейся к атрибутам, с которой мы сталкивались ранее в книге. Однако особый статус, предоставляемый дескрипторам данных, также изменяет смысл *наследования* атрибутов и соответственно смысл имен в вашем коде.

Алгоритм наследования Python: чуть более полная версия

С учетом особого случая дескрипторов данных и общего вызова дескрипторов, разложенного на деревья классов и метаклассов, полный алгоритм наследования нового стиля Python может быть сформулирован в следующем виде. Это сложная процедура, которая предполагает знание дескрипторов, метаклассов и MRO, но все-таки является финальным арбитром при распознавании имен атрибутов (приведенные далее шаги предпринимаются один за другим в соответствии с нумерацией или согласно их порядку слева направо в объединениях "или").

Для поиска явного имени атрибута выполнить описанные ниже шаги.

1. Начиная с *экземпляра* I, провести поиск в экземпляре, в его классе и в суперклассах класса следующим образом.
   а) Искать в словарях `__dict__` всех классов в `__mro__`, найденном в `__class__` экземпляра I.
   б) Если на шаге а) был найден дескриптор данных, тогда вызвать его метод `__get__` и завершить работу.
   в) Иначе возвратить значение в словаре `__dict__` экземпляра I.
   г) Иначе вызвать дескриптор не данных или возвратить значение, найденное на шаге а).
2. Начиная с *класса* C, провести поиск в классе, в его суперклассах и в его дереве метаклассов следующим образом.

а) Искать в словарях __dict__ всех метаклассов в __mro__, найденном в __class__ класса C.

б) Если на шаге а) был найден дескриптор данных, тогда вызвать его метод __get__ и завершить работу.

в) Иначе вызвать дескриптор данных или возвратить значение в словаре __dict__ класса из собственного __mro__ класса C.

г) Иначе вызвать дескриптор не данных или возвратить значение, найденное на шаге а).

3. В шагах 1 и 2 *встроенные* имена по существу используют только источники в пунктах а) (см. далее).

Снова обратите внимание здесь на то, что алгоритм применим только к обычному *явному* извлечению атрибутов. *Неявный* поиск имен методов для *встроенных имен* не соблюдает описанные выше правила и по существу в обоих случаях использует только источники из шагов а), что будет демонстрироваться в следующем разделе.

Как всегда, подразумеваемый суперкласс object предоставляет ряд стандартных методов на верхушке каждого дерева классов и метаклассов (т.е. в конце каждой последовательности MRO). И помимо всего этого может быть запущен метод __getattr__ (если определен), когда атрибут не найден, и метод __getattribute__ для каждой операции извлечения атрибутов, хотя они являются расширениями модели поиска имен, предназначенными для особых случаев. В главе 38 приводилась дополнительная информация об указанных инструментах и дескрипторах, а в главе 32 рассматривался особый случай просмотра MRO для super.

Наследование присваивания

Также обратите внимание, что в предыдущем разделе наследование определялось в терминах *ссылки* на атрибут (поиск), но его части применимы также и к *присваиванию* атрибута. Как уже известно, присваивание обычно изменяет значения атрибутов в самом целевом объекте, но процедура наследования также инициируется с целью первоначальной проверки во время присваивания для ряда инструментов управления атрибутами, рассмотренных в главе 38, включая дескрипторы и свойства. Когда такие инструменты присутствуют, они перехватывают операции присваивания атрибутов и могут произвольно направлять их.

Скажем, при выполнении присваивания атрибутов для классов нового стиля дескриптор данных с методом __set__ получается из класса по наследованию с использованием MRO и имеет приоритет перед нормальной моделью хранения. В переводе на язык правил из предыдущего раздела:

- когда такие операции присваивания применяются к экземпляру, они по существу следуют шагам а)–в) правила 1, выполняя поиск в дереве классов экземпляра, хотя на шаге б) вместо __get__ вызывается __set__, а на шаге в) работа завершается и взамен попытки извлечения производится сохранение в экземпляре;
- когда такие операции присваивания применяются к классу, они запускают такую же процедуру в отношении дерева метаклассов класса: примерно то же самое, что и правило 2, но на шаге в) работа завершается и происходит сохранение в классе.

Поскольку дескрипторы являются основой для других расширенных инструментов управления атрибутами, таких как свойства и слоты, предварительная проверка со стороны процедуры наследования при присваивании задействуется в многочислен-

ных контекстах. Совокупный эффект заключается в том, что в классах нового стиля дескрипторы трактуются как особый случай наследования в случае ссылки *и* присваивания.

Особый случай для встроенных имен

Итак, мы рассмотрели *почти* всю историю. Как выяснилось, встроенные имена не следуют описанным выше правилам. Для встроенных имен экземпляры и классы могут пропускаться, что представляет собой особый случай, который отличается от нормального или явного наследования имен. Из-за того, что это расхождение, *специфичное к контексту*, его легче продемонстрировать в коде, чем вплести в единственный алгоритм. В следующем взаимодействии `str` является встроенным именем, `__str__` — эквивалентом в виде явного имени и экземпляр пропускается только для встроенного имени:

```
>>> class C:                        # Особый случай наследования #2...
        attr = 1                    # Для встроенных имен пропускается шаг
        def __str__(self): return('class')

>>> I = C()
>>> I.__str__(), str(I)             # Оба имени из класса, если нет в экземпляре
('class', 'class')

>>> I.__str__ = lambda: 'instance'
>>> I.__str__(), str(I)             # Явное=>экземпляр, встроенное=>класс!
('instance', 'class')

>>> I.attr                          # Асимметрично с нормальными или явными именами
1
>>> I.attr = 2; I.attr
2
```

Ранее в файле `metaclass5.py` было показано, что то же самое остается верным для *классов*: поиск явных имен начинается с класса, но встроенных — с класса для класса, который является его метаклассом и по умолчанию принимается как `type`:

```
>>> class D(type):
        def __str__(self): return('D class')

>>> class C(D):
        pass
>>> C.__str__(C), str(C)            # Явное=>суперкласс, встроенное=>метакласс!
('D class', "<class '__main__.C'>")

>>> class C(D):
        def __str__(self): return('C class')
>>> C.__str__(C), str(C)            # Явное=>класс, встроенное=>метакласс!
('C class', "<class '__main__.C'>")

>>> class C(metaclass=D):
        def __str__(self): return('C class')
>>> C.__str__(C), str(C)            # Встроенное=>метакласс, определяемый пользователем
('C class', 'D class')
```

На самом деле иногда бывает нелегко узнать, *откуда* поступает имя в такой модели, потому что все классы также наследуются от `object` — в том числе стандартный метакласс `type`. В следующем явном вызове класс `C`, по-видимому, получает стандартный метод `__str__` из `object`, а не из метакласса, согласно первому источнику наследования классов (собственного MRO класса); наоборот, встроенное имя делает пропуск вперед до метакласса, как и ранее:

```
>>> class C(metaclass=D):
        pass
>>> C.__str__(C), str(C)            # Явное=>object, встроенное=>метакласс
("<class '__main__.C'>", 'D class')
>>> C.__str__
<slot wrapper '__str__' of 'object' objects>
>>> for k in (C, C.__class__, type): print([x.__name__ for x in k.__mro__])
['C', 'object']
['D', 'type', 'object']
['type', 'object']
```

Все изложенное подводит нас к финальной цитате из `import this` – принципу, похоже, конфликтующему со статусом, который предоставлен дескрипторам и встроенным именам в механизме наследования атрибутов, относящемся к классам нового стиля:

Особые случаи не настолько особенные, чтобы нарушать правила.

Разумеется, некоторые практические нужды служат основанием для исключений. Мы здесь воздержимся от обоснований, но вы обязаны тщательно обдумать последствия объектно-ориентированного языка, который применяет наследование – *свою фундаментальную операцию* – в такой непрямой и противоречивой манере. По меньшей мере, это должно подчеркнуть важность сохранения вашего кода *простым*, чтобы не делать его зависимым от подобных запутанных правил. Как всегда, пользователи вашего кода и программисты, сопровождающие его, будут только счастливы.

Чтобы получить более достоверные сведения, просмотрите внутреннюю реализацию наследования в Python – полную историю вы найдете в файлах `object.c` и `typeobject.c`, первый из которых предназначен для нормальных экземпляров, а второй для классов. Конечно, использование Python не требует глубокого погружения в его внутреннее устройство, но это основной источник истины в сложной и развивающейся системе, а временами наилучший из того, что вы сумеете найти. Сказанное особенно справедливо в граничных ситуациях, порожденных накопленными исключениями. Давайте теперь перейдем к последнему фрагменту магии, связанной с метаклассами.

Методы метаклассов

Будучи столь же важными, как наследование имен, *методы* в метаклассах обрабатывают их *классы-экземпляры* – не обычные объекты экземпляров, известные нам как `self`, а сами классы. В результате методы метаклассов становятся похожими по духу и форме на *методы классов*, исследованные в главе 32, хотя опять-таки они доступны только в области экземпляров метаклассов, не при нормальном наследовании экземпляров. Например, неудача в конце следующего взаимодействия является результатом действия правил для наследования явных имен из предыдущего раздела:

```
>>> class A(type):
        def x(cls): print('ax', cls)      # Метакласс (экземпляры=классы)
        def y(cls): print('ay', cls)      # y переопределяется экземпляром B
>>> class B(metaclass=A):
        def y(self): print('by', self)    # Нормальный класс
                                          #    (нормальные экземпляры)
        def z(self): print('bz', self)    # Словарь пространств имен хранит y и z
```

```
>>> B.x                          # x получается из метакласса
<bound method A.x of <class '__main__.B'>>
>>> B.y                          # y и z определены в самом классе
<function B.y at 0x0295F1E0>
>>> B.z
<function B.z at 0x0295F378>
>>> B.x()                        # Вызов метода метакласса: получает класс
ax <class '__main__.B'>
>>> I = B()                      # Вызовы методов экземпляра: получают экземпляр
>>> I.y()
by <__main__.B object at 0x02963BE0>
>>> I.z()
bz <__main__.B object at 0x02963BE0>
>>> I.x()                        # Экземпляр не видит имена метакласса
AttributeError: 'B' object has no attribute 'x'
Ошибка атрибута: объект B не имеет атрибута x
```

Методы метаклассов или методы классов

Несмотря на отличие в видимости наследованию, во многом подобно методам классов методы метаклассов предназначены для управления данными уровня классов. На самом деле их роли могут частично совпадать (почти как в целом у метаклассов и декораторов классов), но методы метаклассов доступны исключительно через класс и для привязки к классу не требуют явного объявления `classmethod` на уровне класса. Другими словами, методы метаклассов можно воспринимать как неявные методы классов с ограниченной видимостью:

```
>>> class A(type):
        def a(cls):           # Метод метакласса: получает класс
            cls.x = cls.y + cls.z

>>> class B(metaclass=A):
        y, z = 11, 22
        @classmethod          # Метод класса: получает класс
        def b(cls):
            return cls.x

>>> B.a()          # Вызов метода метакласса; является видимым только классу
>>> B.x            # Создает данные класса в B, доступные нормальным экземплярам
33

>>> I = B()
>>> I.x, I.y, I.z
(33, 11, 22)

>>> I.b()          # Метод класса: передается класс, не экземпляр;
                   #   является видимым экземпляру
33
>>> I.a()          # Методы метакласса: доступны только через класс
AttributeError: 'B' object has no attribute 'a'
Ошибка атрибута: объект B не имеет атрибута a
```

Перегрузка операций в методах метакласса

Точно так же, как нормальные классы, метаклассы могут задействовать перегрузку операций, чтобы сделать встроенные операции применимыми к их классам-экземплярам. Например, метод индексирования `__getitem__` в показанном ниже метаклассе представляет собой метод метакласса, который предназначен для обработки самих

классов – т.е. классов, являющихся экземплярами метакласса, а не экземпляров этих классов, создаваемых впоследствии. Надо сказать, что согласно обрисованным ранее алгоритмам наследования нормальные экземпляры классов вообще не наследуют имена, полученные через отношение экземпляра метакласса, хотя они могут обращаться к именам, присутствующим в их собственных классах:

```
>>> class A(type):
        def __getitem__(cls, i):    # Метод метакласса для обработки классов:
            return cls.data[i]      # Встроенные операции пропускают класс,
                                    #   используют метакласс
                        # Явные имена инициируют поиск в классе и метаклассе
>>> class B(metaclass=A):           # Сначала используются дескрипторы данных
                                    #   в метаклассе
        data = 'spam'

>>> B[0]                    # Имена экземпляра метакласса: видимы только классу
's'
>>> B.__getitem__
<bound method A.__getitem__ of <class '__main__.B'>>

>>> I = B()
>>> I.data, B.data                # Имена, полученные нормальным наследованием:
                                  #   видимы экземпляру и классу
('spam', 'spam')
>>> I[0]
TypeError: 'B' object does not support indexing
Ошибка типа: объект B не поддерживает индексирование
```

В метаклассе также допускается определять метод `__getattr__`, но его можно использовать для обработки только *классов-экземпляров*, а не нормальных экземпляров этих классов – как обычно, он даже не будет получен экземплярами класса:

```
>>> class A(type):
        def __getattr__(cls, name):        # Получается getitem класса B
            return getattr(cls.data, name) # Но не выполняется одинаково
                                           #   для встроенных операций

>>> class B(metaclass=A):
        data = 'spam'

>>> B.upper()
'SPAM'
>>> B.upper
<built-in method upper of str object at 0x029E7420>
>>> B.__getattr__
<bound method A.__getattr__ of <class '__main__.B'>>

>>> I = B()
>>> I.upper
AttributeError: 'B' object has no attribute 'upper'
Ошибка атрибута: объект B не имеет атрибута upper
>>> I.__getattr__
AttributeError: 'B' object has no attribute '__getattr__'
Ошибка атрибута: объект B не имеет атрибута __getattr__
```

Тем не менее, перенос метода `__getattr__` в метакласс не помогает справиться с проблемой отсутствия в нем перехвата встроенных операций. В приведенном далее продолжении явные указываемые атрибуты направляются методу `__getattr__` метакласса, но встроенные операции – нет, вопреки тому факту, что в первом примере

данного раздела операция индексирования направлялась методу __getitem__ метакласса. Это наводит на мысль, что __getattr__ нового стиля является *особым случаем особого случая*, и дальнейшее рекомендуемое упрощение кода избегает зависимости от таких граничных случаев:

```
>>> B.data = [1, 2, 3]
>>> B.append(4) # Явно указанные нормальные имена направляются getattr метакласса
>>> B.data
[1, 2, 3, 4]
>>> B.__getitem__(0)        # Явно указанные особые имена
                            #   направляются getattr метакласса
1
>>> B[0]            # Но встроенные операции тоже пропускают getattr метакласса?!
TypeError: 'A' object does not support indexing
Ошибка типа: объект A не поддерживает индексирование
```

Вероятно, вы уже можете признать, что метаклассы интересно исследовать, но довольно легко утратить представляемую ими общую картину. Ради экономии пространства дополнительные детали здесь не приводятся. Для целей настоящей главы гораздо важнее показать, зачем вообще может возникнуть потребность в применении такого инструмента. Давайте перейдем к нескольким более крупным примерам, которые продемонстрируют роли метаклассов в действии. Как обнаружится, подобно многим инструментам в Python метаклассы в первую очередь направлены на облегчение работы по сопровождению за счет устранения избыточности.

Пример: добавление методов в классы

В этом и следующем разделе мы займемся изучением примеров двух распространенных сценариев использования для метаклассов: добавление методов в класс и автоматическое декорирование всех методов. Они представляют лишь две из многочисленных ролей метаклассов, рассмотреть которые в главе полностью невозможно из-за ее ограниченного объема; более сложные приложения ищите в веб-сети. Однако приводимые далее примеры являются типичными иллюстрациями работы метаклассов, которых вполне достаточно, чтобы пояснить особенности их применения.

Кроме того, они оба дают возможность противопоставить декораторы и метаклассы — в первом примере сравниваются основанные на метаклассах и декораторах реализации дополнения классов и помещения экземпляров в оболочки, а во втором сначала применяется декоратор с метаклассом, а затем с еще одним декоратором. Вы увидите, что оба инструмента часто взаимозаменяемы и даже дополняют друг друга.

Ручное дополнение

Ранее в главе мы взглянули на скелетный код, который дополнял классы за счет добавления к ним методов разнообразными способами. Как выяснилось, простого наследования на основе классов достаточно, если добавочные методы статически известны при реализации класса. Достичь того же самого эффекта часто удается с помощью композиции через внедрение объектов. Тем не менее, для динамических сценариев временами требуются другие методики — обычно хватает вспомогательных функций, но метаклассы обеспечивают явную структуру и минимизируют затраты при сопровождении в будущем.

Давайте воплотим эти идеи в работающем коде. Рассмотрим следующий пример ручного дополнения класса — в нем добавляются два метода к двум классам после того, как они были созданы:

```
# Расширение вручную - добавление новых методов в классы
class Client1:
   def __init__(self, value):
      self.value = value
   def spam(self):
      return self.value * 2

class Client2:
   value = 'ni?'

def eggsfunc(obj):
   return obj.value * 4

def hamfunc(obj, value):
   return value + 'ham'

Client1.eggs = eggsfunc
Client1.ham = hamfunc

Client2.eggs = eggsfunc
Client2.ham = hamfunc

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))
```

Прием работает, потому что методы всегда можно присваивать классу после того, как он был создан, до тех пор, пока присваиваемые методы являются функциями с дополнительным первым аргументом для получения целевого экземпляра `self`. Данный аргумент может использоваться для обращения к информации состояния, доступной из экземпляра класса, хотя функция определяется независимо от класса.

В результате запуска код мы получаем вывод метода, реализованного внутри первого класса, а также двух методов, добавленных в класс после его создания:

```
c:\code> py -3 extend-manual.py
Ni!Ni!
Ni!Ni!Ni!Ni!
baconham
ni?ni?ni?ni?
baconham
```

Такая схема хорошо подходит в отдельных случаях и может применяться для произвольного наполнения класса во время выполнения. Однако она страдает от потенциально значительного недостатка: нам приходится повторять код дополнения для каждого класса, которому нужны новые методы. В нашем случае добавление двух методов к обоим классам было не слишком обременительным, но в более сложных сценариях такой подход может оказаться отнимающим много времени и подверженным ошибкам. Если мы когда-либо забудем сделать это согласованно или возникнет необходимость внести в дополнение какие-то изменения, тогда могут возникнуть проблемы.

Дополнение на основе метаклассов

Несмотря на работоспособность ручного дополнения, в более крупных программах было бы лучше, если бы мы могли применять такие изменения к полному набору классов автоматически. В таком случае мы избежали бы шанса плохо сделать работу для любого отдельно взятого класса. Вдобавок реализация дополнения в единственном месте лучше поддерживает будущие изменения — все классы будут подхватывать изменения автоматически.

Один из способов достижения указанной цели предусматривает использование метаклассов. Если мы реализуем дополнение в метаклассе, тогда каждый класс, который объявляет этот метакласс, будет единообразно и корректно дополняться и автоматически подхватывать любые будущие изменения. Сказанное демонстрируется в следующем коде:

```
# Расширение с помощью метакласса - лучше поддерживает будущие изменения

def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'ham'

class Extender(type):
    def __new__(meta, classname, supers, classdict):
        classdict['eggs'] = eggsfunc
        classdict['ham'] = hamfunc
        return type.__new__(meta, classname, supers, classdict)

class Client1(metaclass=Extender):
    def __init__(self, value):
        self.value = value
    def spam(self):
        return self.value * 2

class Client2(metaclass=Extender):
    value = 'ni?'

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))
```

Теперь оба клиентских класса расширяются новыми методами, поскольку они являются экземплярами метакласса, который выполняет дополнение. Запуск данной версии дает такой же вывод, как и ранее — мы не изменяли то, что делает код, а всего лишь провели его рефакторинг с целью более аккуратной инкапсуляции дополнения:

```
c:\code> py -3 extend-meta.py
Ni!Ni!
Ni!Ni!Ni!Ni!
baconham
ni?ni?ni?ni?
baconham
```

Обратите внимание, что метакласс в приведенном примере по-прежнему решает довольно статичную задачу: добавление двух известных методов к каждому классу, который объявляет метакласс. В сущности, если все, что нам нужно — это всегда добавлять те же самые два метода к набору классов, то мы могли бы также реализовать их в обычном суперклассе и наследовать его в подклассах. Но на практике структура на базе метаклассов поддерживает более динамичное поведение. Например, целевой класс также можно было бы конфигурировать на основе произвольной логики во время выполнения:

```
# Класс можно также конфигурировать на основе проверок во время выполнения
class MetaExtend(type):
   def __new__(meta, classname, supers, classdict):
      if sometest():
         classdict['eggs'] = eggsfunc1
      else:
         classdict['eggs'] = eggsfunc2
      if someothertest():
         classdict['ham'] = hamfunc
      else:
         classdict['ham'] = lambda *args: 'Not supported'
                                                   # Не поддерживается
      return type.__new__(meta, classname, supers, classdict)
```

Метаклассы против декораторов классов: раунд 2

Запомните еще раз: в плане функциональности декораторы классов из предыдущего раздела часто пересекаются с метаклассами, обсуждаемыми в настоящей главе. Это происходит из того факта, что:

- декораторы классов повторно привязывают имена классов к результату функции в конце оператора `class` после того, как класс был создан;
- метаклассы работают путем прогона процедуры создания объектов классов через объект в конце оператора `class`, чтобы создать новый класс.

Несмотря на то что модели несколько отличаются, на деле они нередко могут достигать одинаковых целей, хотя и разными способами. Как вы теперь видите, декораторы классов напрямую соответствуют методам `__init__` метаклассов, вызываемым для инициализации вновь созданных классов. Декораторы не имеют прямого аналога для методов `__new__` метаклассов (вызываемых в первую очередь для создания классов) или других методов метаклассов (применяемых для обработки классов-экземпляров), но многие или большинство сценариев использования для таких инструментов не требуют этих дополнительных шагов.

По указанным причинам оба инструмента в принципе могут применяться для управления экземплярами класса и самим классом. Тем не менее, на практике метаклассы влекут за собой добавочные шаги для управления экземплярами, а декораторы — добавочные шаги для создания новых классов. Следовательно, наряду с тем, что их роли часто пересекаются, метаклассы лучше всего использовать для управления объектами классов. Давайте займемся воплощением изложенных идей в коде.

Дополнение на основе декораторов

В случаях чистого дополнения декораторы нередко способны заменять метаклассы. Скажем, пример метакласса из предыдущего раздела, который добавлял методы в

класс при его создании, можно было бы реализовать также в виде декоратора класса; в таком режиме декораторы грубо соответствуют методу `__init__` метаклассов, потому что к моменту вызова декоратора объект класса уже был создан. Как и для метаклассов, исходный тип класса предохраняется, поскольку уровень оболочки не вставляется. Вывод, полученный в результате запуска файла extend-deco.py со следующим содержимым, будет таким же, как у ранее показанного кода метакласса:

```
# Расширение с помощью декоратора: то же самое, что и предоставление
#  метода __init__ в метаклассе

def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'ham'

def Extender(aClass):
    aClass.eggs = eggsfunc              # Управляет классом, не экземпляром
    aClass.ham = hamfunc                # Эквивалентно методу __init__ метакласса
    return aClass

@Extender
class Client1:                          # Client1 = Extender(Client1)
    def __init__(self, value):          # Повторная привязка в конце оператора class
        self.value = value
    def spam(self):
        return self.value * 2

@Extender
class Client2:
    value = 'ni?'

X = Client1('Ni!')                      # X - экземпляр Client1
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))
```

Другими словами, по крайней мере, в определенных случаях, декораторы способны управлять классами так же легко, как метаклассы. Однако обратное не настолько прямолинейно; метаклассы могут применяться для управления экземплярами, но лишь за счет написания некоторого объема дополнительной логики, что и демонстрируется в следующем разделе.

Управление экземплярами вместо классов

Как только что было показано, декораторы классов часто могут выступать в той же роли *управления классами*, как и метаклассы. Метаклассы нередко способны исполнять ту же роль *управления экземплярами*, что и декораторы, но это требует добавочного кода и может выглядеть менее естественным. То есть:

- декораторы классов могут управлять классами и экземплярами, но не создавать классы обычным образом;
- метаклассы могут управлять классами и экземплярами, но для экземпляров требуется дополнительная работа.

Тем не менее, определенные приложения может быть эффективнее реализовывать с помощью первого или второго инструмента. Скажем, рассмотрим приведенный ниже пример декоратора классов, взятый из предыдущей главы; он используется для вывода трассировочного сообщения всякий раз, когда извлекается нормально именованный атрибут экземпляра класса:

```
# Декоратор классов для трассировки внешних операций,
#   извлекающих атрибуты экземпляров

def Tracer(aClass):                                # При декорировании @
    class Wrapper:
        def __init__(self, *args, **kargs):        # При создании экземпляров
            self.wrapped = aClass(*args, **kargs)  # Использование имени из
                                                   # объемлющей области видимости
        def __getattr__(self, attrname):
            print('Trace:', attrname)  # Перехват всех атрибутов кроме .wrapped
            return getattr(self.wrapped, attrname) # Делегирование внутреннему
                                                   #   объекту
    return Wrapper

@Tracer
class Person:                                      # Person = Tracer(Person)
    def __init__(self, name, hours, rate):         # Wrapper запоминает Person
        self.name = name
        self.hours = hours
        self.rate = rate        # Операции доступа внутри методов не отслеживаются
    def pay(self):
        return self.hours * self.rate

bob = Person('Bob', 40, 50)    # bob - на самом деле экземпляр Wrapper
print(bob.name)                # Wrapper содержит внедренный экземпляр Person
print(bob.pay())               # Запускается __getattr__
```

После запуска кода декоратор применяет повторную привязку имени класса для помещения объектов экземпляра внутрь объекта, который выдает трассировочные сообщения в следующем выводе:

```
c:\code> py -3 manage-inst-deco.py
Trace: name
Bob
Trace: pay
2000
```

Хотя метакласс способен достичь того же эффекта, концептуально он выглядит менее прямолинейным. Метаклассы явно спроектированы для управления созданием объектов классов и обладают интерфейсом, приспособленным для этой цели. Чтобы использовать метакласс именно для управления экземплярами, мы также обязаны взять на себя ответственность и за создание класса — избыточный шаг, если в противном случае было бы достаточно нормальной процедуры создания класса. Показанный ниже метакласс (файл `manage-inst-meta.py`) дает тот же самый эффект, что и предыдущий декоратор:

```
# Управление экземплярами как в предыдущем примере, но с помощью метакласса

def Tracer(classname, supers, classdict):      # При вызове, создающем класс
    aClass = type(classname, supers, classdict) # Создание клиентского класса
```

```
    class Wrapper:
        def __init__(self, *args, **kargs):      # При создании экземпляров
            self.wrapped = aClass(*args, **kargs)
        def __getattr__(self, attrname):
            print('Trace:', attrname)             # Перехват всех атрибутов
                                                  #   кроме .wrapped
            return getattr(self.wrapped, attrname) # Делегирование внутреннему
                                                   #   объекту
    return Wrapper

class Person(metaclass=Tracer):                    # Создание Person с Tracer
    def __init__(self, name, hours, rate):         # Wrapper запоминает Person
        self.name = name
        self.hours = hours
        self.rate = rate        # Извлечение внутри методов не отслеживается
    def pay(self):
        return self.hours * self.rate

bob = Person('Bob', 40, 50)    # bob - на самом деле экземпляр Wrapper
print(bob.name)                # Wrapper содержит внедренный экземпляр Person
print(bob.pay())               # Запускается __getattr__
```

Код работает, но полагается на две уловки. Во-первых, в нем должна применяться простая функция вместо класса, потому что подклассы `type` обязаны придерживаться протоколов создания объектов. Во-вторых, в нем должен вручную создаваться целевой класс обращением к `type`; вообще-то необходимо возвращать оболочку экземпляра, но метаклассы также ответственны за создание и возвращение целевого класса. По правде говоря, в этом примере мы использовали протокол метаклассов для имитирования декораторов, а не наоборот; поскольку оба инструмента запускаются при завершении оператора `class`, во многих ролях они являются всего лишь вариациями на тему. Версия с метаклассом выдает тот же самый вывод, что и версия с декоратором:

```
c:\code> py -3 manage-inst-meta.py
Trace: name
Bob
Trace: pay
2000
```

Вы должны самостоятельно исследовать обе версии примеров, чтобы оценить связанные с ними компромиссы. Однако в целом метаклассы вероятно лучше подходят для управления классами, потому что так они были спроектированы; декораторы классов способны управлять либо экземплярами, либо классами, хотя могут оказаться не наилучшим выбором для исполнения более сложных ролей, которые в настоящей книге не раскрываются. Дополнительные примеры применения метаклассов ищите в веб-сети, но имейте в виду, что одни могут быть более подходящими, чем другие (а некоторые из их авторов могут знать Python меньше, чем вы!).

Эквивалентность метаклассов и декораторов классов?

В предыдущем разделе выяснилось, что при использовании для управления экземплярами метаклассы требуют добавочного шага по созданию класса и потому не могут полностью замещать декораторы во всех сценариях применения. Но как насчет обратного: являются ли декораторы заменой для метаклассов?

На тот случай, если книга еще взорвала вам мозг, взгляните также на следующую альтернативную реализацию — декоратор классов, который возвращает экземпляр метакласса:

```
# Декоратор может обращаться к метаклассу, хотя не наоборот без type()
>>> class Metaclass(type):
        def __new__(meta, clsname, supers, attrdict):
            print('In M.__new__:')
            print([clsname, supers, list(attrdict.keys())])
            return type.__new__(meta, clsname, supers, attrdict)

>>> def decorator(cls):
        return Metaclass(cls.__name__, cls.__bases__, dict(cls.__dict__))

>>> class A:
        x = 1

>>> @decorator
    class B(A):
        y = 2
        def m(self): return self.x + self.y

In M.__new__:
['B', (<class '__main__.A'>,), ['__qualname__', '__doc__', 'm', 'y',
'__module__']]
>>> B.x, B.y
(1, 2)
>>> I = B()
>>> I.x, I.y, I.m()
(1, 2, 3)
```

Это почти доказывает эквивалентность двух инструментов, но на самом деле только в терминах *координирования* во время создания класса. Декораторы снова исполняют те же роли, что и методы `__init__` метаклассов. Поскольку данный декоратор возвращает экземпляр метакласса, здесь по-прежнему предполагаются метаклассы или, во всяком случае, их суперкласс `type`. Кроме того, после создания класса инициируется *дополнительное* обращение к метаклассу и такая схема не является идеальной в реальном коде — вы также могли бы перенести данный метакласс в первый шаг создания:

```
>>> class B(A, metaclass=Metaclass): ...     # Тот же самый эффект, но
                                             #   создает только один класс
```

Тем не менее, здесь присутствует некоторая избыточность инструментов, а роли декораторов и метаклассов на практике нередко пересекаются. И хотя декораторы напрямую не поддерживают понятие методов уровня классов в обсуждаемых ранее метаклассах, похожие результаты можно получить с помощью методов и состояния в *объектах-посредниках*, создаваемых декораторами, но последнее наблюдение мы оставляем для самостоятельного исследования.

Обратное может не выглядеть применимым — метакласс в целом нельзя отсылать декоратору, отличающемуся от метакласса, потому что до тех пор, пока обращение к метаклассу не завершится, класс еще не существует — хотя метакласс *может* принимать форму простого вызываемого объекта, который запускает `type` для создания класса напрямую и его передачи декоратору. Другими словами, решающей привязкой в такой модели является вызов `type`, выданный для создания класса. С учетом этого метаклассы и декораторы классов часто функционально эквивалентны с варьирующимися моделями *протокола координирования*:

```
>>> def Metaclass(clsname, supers, attrdict):
        return decorator(type(clsname, supers, attrdict))
```

```
>>> def decorator(cls): ...
>>> class B(A, metaclass=Metaclass): ...     # Метаклассы могут обращаться
                                            #   к декораторам и наоборот
```

В действительности метаклассы не обязательно должны возвращать экземпляр `type` — подойдет *любой* объект, согласующийся с ожиданиями разработчика класса – и это еще больше размывает отличие между декораторами и метаклассами:

```
>>> def func(name, supers, attrs):
        return 'spam'

>>> class C(metaclass=func):  # Класс, чей метакласс делает его строкой!
        attr = 'huh?'

>>> C, C.upper()
('spam', 'SPAM')

>>> def func(cls):
        return 'spam'

>>> @func
    class C:                   # Класс, чей декоратор делает его строкой!
        attr = 'huh?'

>>> C, C.upper()
('spam', 'SPAM')
```

Оставив в стороне трюки подобного рода с метаклассами и декораторами, на практике их роли часто определяются временем, как было указано ранее.

- Поскольку декораторы запускаются после создания класса, в ролях с созданием классов они влекут за собой дополнительный шаг во время выполнения.
- Поскольку метаклассы должны создавать классы, в ролях с управлением экземплярами они влекут за собой дополнительный шаг на этапе реализации.

В итоге ни один из инструментов полностью не заменяет другой. Строго говоря, метаклассы могут быть функциональным надмножеством, т.к. они способны обращаться к декораторам во время создания классов, но метаклассы также могут оказаться существенно сложнее для понимания и реализации, а многие роли совпадают полностью. На поверку необходимость возложить на себя весь процесс создания классов вероятно гораздо менее важна, чем внедрение в сам процесс.

Однако вместо того, чтобы ползти дальше по этой кроличьей норе, давайте перейдем к исследованию ролей метаклассов, которые могут быть более типичными и практичными. Следующий раздел оканчивает главу еще одним распространенным сценарием использования – автоматическое применение операций к методам класса во время создания классов.

Пример: применение декораторов к методам

Как было показано в предыдущем разделе, поскольку метаклассы и декораторы запускаются в конце оператора `class`, они часто могут применяться *взаимозаменяемо*, несмотря на разный синтаксис. Выбор между двумя инструментами во многих контекстах произволен. Их также можно использовать в *комбинации* как дополняющие друг друга инструменты. В этом разделе мы исследуем пример именно такой комбинации – применение декоратора функций ко всем методам класса.

Трассировка с помощью декорирования вручную

В предыдущей главе мы реализовали два декоратора функций – первый трассировал и подсчитывал вызовы декорированной функции, а второй измерял время выполнения таких вызовов. Там они принимали разнообразные формы, часть которых были применимы к функциям и методам, а часть нет. Мы собрали финальные формы обоих декораторов в файл модуля с целью многократного использования:

```
# Файл decotools.py: смешанные декораторные инструменты
import time

def tracer(func):     # Использовать функцию, а не класс с методом __call__
    calls = 0         # Иначе self - только экземпляр декоратора
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall

def timer(label='', trace=True):   # При наличии аргументов декоратора:
                                   #   предохранить аргументы
    def onDecorator(func):         # При синтаксисе @: предохранить
                                   #   декорированную функцию
        def onCall(*args, **kargs): # При вызовах: вызвать исходную функцию
            start = time.clock()   # Состоянием являются области видимости
                                   #   и атрибут функции
            result = func(*args, **kargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            if trace:
                format = '%s%s: %.5f, %.5f'
                values = (label, func.__name__, elapsed, onCall.alltime)
                print(format % values)
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator
```

Как выяснилось в предыдущей главе, для применения таких декораторов вручную мы просто импортируем их из модуля и записываем код декорирования @ перед каждым методом, для которого необходима трассировка или измерение времени:

```
from decotools import tracer

class Person:
    @tracer
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer
    def giveRaise(self, percent):        # giveRaise = tracer(giverRaise)
        self.pay *= (1.0 + percent)      # onCall запоминает giveRaise

    @tracer
    def lastName(self):                  # lastName = tracer(lastName)
        return self.name.split()[-1]
```

```
bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)                     # Запускается onCall(sue, .10)
print('%.2f' % sue.pay)
print(bob.lastName(), sue.lastName())  # Запускается onCall(bob),
                                       #   запоминается lastName
```

Запуск кода приводит к получению следующего вывода — вызовы декорированных методов направляются логике, которая перехватывает и затем делегирует их выполнение, т.к. имена исходных методов были привязаны к декоратору:

```
c:\code> py -3 decoall-manual.py
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.00
call 1 to lastName
call 2 to lastName
Smith Jones
```

Трассировка с помощью метаклассов и декораторов

Схема с ручным декорированием из предыдущего раздела работоспособна, но требует от нас добавлять синтаксис декорирования перед *каждым* методом, подлежащим трассировке, и позже удалять его, когда его трассировка больше не нужна. Если мы хотим трассировать все методы класса, то в крупных программах это может стать утомительным. В более динамичных контекстах, где дополнение зависит от параметров времени выполнения, схема с ручным декорированием может вообще оказаться неприемлемой. Было бы лучше, если бы мы каким-то образом могли применять декоратор трассировки ко всем методам класса автоматически.

Именно такой прием возможен благодаря метаклассам — поскольку они запускаются при создании класса, то становятся естественным средством для добавления декорирующих оболочек к методам класса. Просматривая словарь атрибутов класса и проверяя их на предмет принадлежности к объектам функций, мы можем автоматически прогонять методы через декоратор и повторно привязывать исходные имена к результатам. Эффект будет каким же, как автоматическая привязка имен методов к декораторам, но мы можем применять его более глобально:

```
# Метакласс, который добавляет декоратор трассировки к каждому методу
#   клиентского класса

from types import FunctionType
from decotools import tracer

class MetaTrace(type):
    def __new__(meta, classname, supers, classdict):
        for attr, attrval in classdict.items():
            if type(attrval) is FunctionType:                # Метод?
                classdict[attr] = tracer(attrval)            # Декорировать его
        return type.__new__(meta, classname, supers, classdict)  # Создать класс

class Person(metaclass=MetaTrace):
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
```

```
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print('%.2f' % sue.pay)
print(bob.lastName(), sue.lastName())
```

Запуск кода приводит к получению тех же результатов, что и ранее — вызовы методов направляются сначала декоратору трассировки для отслеживания и затем передаются исходному методу:

```
c:\code> py -3 decoall-meta.py
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.00
call 1 to lastName
call 2 to lastName
Smith Jones
```

Итог, который вы здесь видите, является *комбинацией* работы декоратора и метакласса — метакласс автоматически применяет декоратор функций к каждому методу во время создания класса, а декоратор функций автоматически перехватывает вызовы методов для того, чтобы выводить трассировочные сообщения. Комбинация "просто работает" благодаря универсальности обоих инструментов.

Применение любого декоратора к методам

Предыдущий пример метакласса имел дело только с одним конкретным декоратором функций — декоратором трассировки. Тем не менее, его несложно обобщить для применения *любого* декоратора ко всем методам класса. Все, что нам понадобится сделать — это добавить внешнюю область видимости, чтобы предохранить желаемый декоратор, почти как мы поступали с декораторами в предыдущей главе. Ниже демонстрируется такое обобщение, которое используется для применения декоратора трассировки еще раз:

```
# Фабрика метаклассов: применение любого декоратора ко всем методам класса

from types import FunctionType
from decotools import tracer, timer

def decorateAll(decorator):
    class MetaDecorate(type):
        def __new__(meta, classname, supers, classdict):
            for attr, attrval in classdict.items():
                if type(attrval) is FunctionType:
                    classdict[attr] = decorator(attrval)
            return type.__new__(meta, classname, supers, classdict)
    return MetaDecorate

class Person(metaclass=decorateAll(tracer)):     # Применение декоратора
                                                 #   ко всем методам
```

```
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print('%.2f' % sue.pay)
print(bob.lastName(), sue.lastName())
```

В результате запуска кода в том виде, как есть, снова получается тот же вывод, что и в предшествующих примерах. Мы по-прежнему в конечном итоге декорируем каждый метод в клиентском классе посредством декоратора функций, но делаем это в более обобщенной манере:

```
c:\code> py -3 decoall-meta-any.py
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.00
call 1 to lastName
call 2 to lastName
Smith Jones
```

Теперь для применения к методам *другого* декоратора мы можем просто заменить имя декоратора в строке заголовка оператора `class`. Например, чтобы задействовать реализованный ранее декоратор функций для измерения времени, мы могли бы при определении класса использовать любую из последних двух строк заголовка, показанных ниже — первая принимает аргументы со стандартными значениями, а вторая задает текст метки:

```
class Person(metaclass=decorateAll(tracer)):      # Применение tracer

class Person(metaclass=decorateAll(timer())):     # Применение timer,
                                                  #   стандартные значения
class Person(metaclass=decorateAll(timer(label='**'))): # Аргументы декоратора
```

Обратите внимание, что представленная схема не способна поддерживать аргументы декоратора, не имеющие стандартных значений, которые отличаются для каждого метода в клиентском классе, но можно передавать аргументы декоратора, применяемые ко всем таким методам, как здесь было сделано. Для тестирования используйте последнее из этих объявлений метаклассов, чтобы применить декоратор `timer`, и добавьте следующие строки в конец сценария с целью вывода дополнительных информационных атрибутов:

```
# Если используется timer: суммарное время для каждого метода

print('-'*40)
print('%.5f' % Person.__init__.alltime)
print('%.5f' % Person.giveRaise.alltime)
print('%.5f' % Person.lastName.alltime)
```

Ниже приведен новый вывод – теперь метакласс помещает методы в оболочку декоратора измерения времени, так что мы можем видеть, сколько времени отнимает каждый вызов, для каждого метода класса:

```
c:\code> py -3 decoall-meta-any2.py
**__init__: 0.00001, 0.00001
**__init__: 0.00001, 0.00001
Bob Smith Sue Jones
**giveRaise: 0.00002, 0.00002
110000.00
**lastName: 0.00002, 0.00002
**lastName: 0.00002, 0.00004
Smith Jones
----------------------------------------
0.00001
0.00002
0.00004
```

Метаклассы против декораторов классов: раунд 3 (и последний)

Как и следовало ожидать, декораторы классов здесь тоже пересекаются с метаклассами. В показанной далее версии метакласс из предыдущего примера заменяется декоратором классов. То есть в ней определяется и используется *декоратор классов, который применяет декоратор функций* ко всем методам класса. Хотя предшествующее высказывание может быть больше похоже на цитату из дзен-буддизма, чем на техническое описание, все работает вполне естественно – декораторы Python поддерживают произвольное вложение и сочетание:

```
# Фабрика декораторов классов: применение любого декоратора ко всем методам класса

from types import FunctionType
from decotools import tracer, timer

def decorateAll(decorator):
    def DecoDecorate(aClass):
        for attr, attrval in aClass.__dict__.items():
            if type(attrval) is FunctionType:
                setattr(aClass, attr, decorator(attrval))    # Не __dict__
        return aClass
    return DecoDecorate

@decorateAll(tracer)                 # Использование декоратора классов
class Person:                        # Применение декоратора функций к методам
    def __init__(self, name, pay):   # Person = decorateAll(..)(Person)
        self.name = name             # Person = DecoDecorate(Person)
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print('%.2f' % sue.pay)
print(bob.lastName(), sue.lastName())
```

Когда код выполняется в имеющемся виде, декоратор классов применяет декоратор функций для трассировки к каждому методу и при их вызовах выводит трассировочные сообщения (вывод получается такой же, как у версии с метаклассом):

```
c:\code> py -3 decoall-deco-any.py
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.00
call 1 to lastName
call 2 to lastName
Smith Jones
```

Обратите внимание, что декоратор класса возвращает исходный дополненный класс, а не оболочку для него (что обычно бывает при помещении внутрь нее объектов экземпляров). Как и в версии с метаклассом, мы предохраняем тип исходного класса – экземпляр Person является экземпляром Person, а не какого-то класса оболочки. Фактически данный декоратор классов имеет дело только с созданием класса; вызовы, создающие экземпляры, вообще не перехватываются.

Такое различие может иметь значение в программах, которые требуют проверки типов для экземпляров, чтобы выдавать исходный класс, а не оболочку. При дополнении класса вместо экземпляра декораторы классов могут предохранять тип исходного класса. Методы класса не являются его исходными функциями, потому что они повторно привязаны к декораторам, но это вероятно менее важно на практике и также верно в альтернативной версии с метаклассом.

Также обратите внимание, что подобно версии с метаклассом имеющаяся структура не способна поддерживать аргументы декоратора функций, которые отличаются для каждого метода в декорированном классе, но может обрабатывать такие аргументы, если они применяются ко всем методам. Чтобы использовать эту схему для применения, например, декоратора измерения времени, будет достаточно одной из двух последних строк декорирования, показанных ниже, если она находится прямо перед определением нашего класса – в первой используются аргументы декоратора со стандартными значениями, а во второй один аргумент указывается явно:

```
@decorateAll(tracer)        # Декорировать все методы с помощью tracer

@decorateAll(timer())       # Декорировать все методы с помощью timer,
                            #   стандартные значения
@decorateAll(timer(label='@@')) # То же, но с передачей аргумента декоратора
```

Как и ранее, воспользуйтесь последней из строк декорирования и добавьте следующий код в конец сценария, чтобы протестировать пример с другим декоратором (разумеется, здесь возможны более эффективные схемы тестирования, но мы уже приблизились к концу главы; при желании можете внести улучшения):

```
# Если используется timer: суммарное время для каждого метода

print('-'*40)
print('%.5f' % Person.__init__.alltime)
print('%.5f' % Person.giveRaise.alltime)
print('%.5f' % Person.lastName.alltime)
```

Появляется вывод того же самого вида – для каждого метода получается время выполнения каждого и всех его вызовов, но декоратору измерения времени был передан другой аргумент метки:

```
c:\code> py -3 decoall-deco-any2.py
@@__init__: 0.00001, 0.00001
@@__init__: 0.00001, 0.00001
Bob Smith Sue Jones
@@giveRaise: 0.00002, 0.00002
110000.00
@@lastName: 0.00002, 0.00002
@@lastName: 0.00002, 0.00004
Smith Jones
----------------------------------------
0.00001
0.00002
0.00004
```

Наконец, декораторы можно объединять, так что каждый будет запускаться для каждого вызова метода, но вероятно потребуется внести изменения в имеющиеся реализации. В таком виде непосредственное вложение их вызовов приводит к трассировке или измерению времени создания при применении другого декоратора. Указание их двух в отдельных строках дает в результате трассировку или измерение времени выполнения для оболочки другого декоратора перед запуском исходного метода. В принципе, метаклассы в этом отношении выглядят не лучше.

```
@decorateAll(tracer(timer(label='@@')))  # Трассирует применение декоратора timer
class Person:

@decorateAll(tracer)                     # Трассирует оболочку onCall, измеряет
                                         #   время выполнения методов
@decorateAll(timer(label='@@'))
class Person:

@decorateAll(timer(label='@@'))
@decorateAll(tracer)                     # Измеряет время выполнения оболочки
                                         #   onCall, трассирует методы
class Person:
```

Дальнейшие размышления на данную тему вы должны продолжить самостоятельно — в книге на это просто не хватает места.

Как видите, метаклассы и декораторы классов не только часто оказываются взаимозаменяемыми, но также обычно дополняют друг друга. Оба инструмента предлагают сложные, но мощные способы настройки и управления классами и объектами экземпляров, поскольку оба, в конечном счете, дают возможность вставлять код в процесс создания классов. Хотя некоторые более продвинутые приложения могут быть более эффективно реализованы с помощью того или другого инструмента, выбор какого-то одного или же комбинации двух инструментов в значительной степени зависит от вас.

Резюме

В главе мы обсуждали метаклассы и исследовали примеры их использования. Метаклассы позволяют внедряться в протокол создания классов Python с целью управления или дополнения определяемых пользователем классов. Из-за того, что метаклассы автоматизируют этот процесс, они могут обеспечивать для разработчиков API-интерфейсов более удачные решения, нежели написанный вручную код или вспомогательные функции; поскольку метаклассы инкапсулируют такой код, они способны минимизировать затраты на сопровождение эффективнее других подходов.

Попутно мы также выяснили, что роли декораторов классов и метаклассов часто пересекаются: так как оба инструмента запускаются в конце оператора `class`, временами они могут применяться взаимозаменяемо. Декораторы классов и метаклассы можно использоваться для управления классами и объектами экземпляров, хотя в ряде сценариев применения с каждым инструментом могут быть связаны компромиссы.

Поскольку в текущей главе раскрывалась сложная тема, мы ограничимся лишь несколькими контрольными вопросами для закрепления основ (откровенно говоря, если вы зашли настолько далеко в главе, посвященной метаклассам, то вероятно уже заслуживаете уважения!). Учитывая тот факт, что эта часть является последней в книге, мы отказываемся от упражнений, обычно приводимых в конце частей. Обязательно ознакомьтесь с приложениями, в которых описаны изменения Python, решения упражнений из предшествующих частей и т.д.; в решениях упражнений вы найдете образцы типичных прикладных программ для самостоятельного изучения.

Завершив отвечать на контрольные вопросы, вы официально доберетесь до конца технического материала настоящей книги. В следующей — финальной — главе предлагается несколько кратких заключительных мыслей, чтобы подвести итоги по книге в целом. После того, как вы проработаете последние контрольные вопросы, жду встречи с вами в благословенном мире Python.

Проверьте свои знания: контрольные вопросы

1. Что такое метакласс?
2. Как объявить метакласс класса?
3. Каким образом декораторы классов пересекаются с метаклассами при управлении классами?
4. Каким образом декораторы классов пересекаются с метаклассами при управлении экземплярами?
5. Что бы вы предпочли иметь среди своего оружия — декораторы или метаклассы? (Пожалуйста, сформулируйте свой ответ в стиле популярного скетча "Испанская инквизиция".)

Проверьте свои знания: ответы

1. Метакласс — это класс, используемый для создания класса. Нормальные классы нового стиля по умолчанию являются экземплярами класса `type`. Метаклассы обычно представляют собой подклассы класса `type`, которые переопределяют методы протокола создания классов, чтобы настраивать вызов создания класса, выдаваемый в конце оператора `class`; как правило, они переопределяют методы `__new__` и `__init__` для внедрения в протокол создания классов. Метаклассы можно реализовывать и другими способами (скажем, как простые функции), но они всегда несут ответственность за создание и возвращение объекта для нового класса. Метаклассы также могут иметь методы и данные, чтобы снабжать линией поведения свои классы — и основывать второе направление поиска при наследовании, — но атрибуты метаклассов доступны только их классам-экземплярам, но не экземплярам этих классов-экземпляров.
2. В Python 3.X используйте ключевой аргумент в строке заголовка оператора `class`: `class C(metaclass=M)`. В Python 2.X взамен применяйте атрибут клас-

са: `__metaclass__ = M`. В Python 3.X в строке заголовка оператора `class` перед ключевым аргументом `metaclass` могут также указываться имена нормальных суперклассов; кроме того, в Python 2.X обычно вы должны явно наследовать от `object`, хотя иногда это необязательно.

3. Поскольку декораторы классов и метаклассы автоматически запускаются в конце оператора `class`, оба инструмента могут использоваться для управления классами. Декораторы повторно привязывают имя класса к результату, полученному от вызываемого объекта, а метаклассы прогоняют процедуру создания класса через вызываемый объект, но обе привязки применяются для похожих целей. Чтобы управлять классами, декораторы просто дополняют и возвращают объекты исходных классов. Метаклассы дополняют класс после того, как они создали его. Декораторы в такой роли могут обладать небольшим недостатком, если должен определяться новый класс, потому что исходный класс уже был создан.

4. Из-за того, что декораторы классов и метаклассы автоматически запускаются в конце оператора `class`, мы можем использовать оба инструмента для управления экземплярами классов путем вставки объекта оболочки (посредника), который перехватывает вызовы, создающие экземпляр. Декораторы могут повторно привязывать имя класса к вызываемому объекту, запускаемому при создании экземпляров, который предохраняет объект исходного класса. Метаклассы способны делать то же самое, но могут обладать небольшим недостатком в такой роли, потому что они обязаны также создавать объект класса.

5. Наше главное оружие – декораторы... декораторы и метаклассы... метаклассы и декораторы... Два наших оружия – метаклассы и декораторы... и безжалостная эффективность... *Три* наших оружия – метаклассы, декораторы и безжалостная эффективность... и почти фанатичная преданность Python... *Четыре* наших... О нет... *Среди* наших оружий... Среди нашего оружия... есть такие элементы, как метаклассы, декораторы ... Я вернусь.

ГЛАВА 41

Все хорошее когда-нибудь заканчивается

Встречайте окончание книги! Теперь, когда вы продвинулись настолько далеко, в заключение я хочу сказать несколько слов об эволюции Python, прежде чем отпускать вас в мир разработки программного обеспечения. Конечно, эта тема субъективна по своей природе, однако жизненно важна для всех пользователей Python.

У вас была возможность самостоятельно увидеть целиком весь язык — включая ряд расширенных средств, которые могут выглядеть расходящимися с его парадигмой написания сценариев. Хотя многие с пониманием воспримут это как статус-кво, в проекте с открытым кодом очень важно, чтобы некоторые задавали также и вопросы “почему”. Ведь в итоге путь истории Python — и ее истинный результат — по крайней мере, частично зависит от вас.

Парадокс Python

Если вы читали настоящую книгу или подходящее ее подмножество, тогда должны быть в состоянии беспристрастно взвешивать компромиссы, касающиеся Python. Вы видели, что Python представляет собой мощный, выразительный и даже забавный язык программирования, который послужит технологией, позволяющей вам двигаться дальше куда угодно. Одновременно вы также заметили, что современный Python является чем-то вроде парадокса: он расширяется, чтобы содержать инструменты, которые многие считают совершенно избыточными и необычайно сложными — со скоростью, которая, по-видимому, только увеличивается.

Со своей стороны, как один из первых сторонников Python, я наблюдал, как с годами он трансформировался из простого средства в сложно устроенный инструмент с постоянно меняющимися возможностями. Похоже, его сложность возросла, по крайней мере, до уровня сложности других языков, из-за чего многие из нас перешли на Python. И точно так же, как в других языках, ситуация неизбежно благоприятствовала растущей культуре, где непонятность считается признаком славы.

Сложившееся положение противоречит первоначальным целям Python настолько, насколько вообще возможно. Запустите команду `import this` в любом интерактивном сеансе Python, чтобы увидеть, что я имею в виду — мировоззрение, из которого я неоднократно приводил вытяжки в контекстах, где оно очевидно нарушалось. На многих уровнях его основные идеалы ясности, простоты и отсутствия избыточности были либо простодушно забыты, либо легкомысленно отброшены.

Конечным результатом оказывается язык и сообщество, которые в наши дни можно было бы описать рядом терминов, используемых мною в отношении языка Perl в главе 1 первого тома. Несмотря на то что Python все еще может многое предложить, как объясняется в следующем разделе, такая тенденция грозит свести на нет большую часть его осознаваемого преимущества.

О "необязательных" языковых средствах

Ближе к началу предыдущей главы я приводил мнение Тима Петерса о том, что метаклассы не являются предметом интереса для 99% программистов на Python, чтобы подчеркнуть их кажущуюся непонятность. Тем не менее, утверждение не вполне точно, причем не только в числовом отношении. Автор мнения – известный участник процесса разработки и сторонник языка Python с первых дней его существования, и я не намеревался выбирать кого-нибудь просто так. Кроме того, я и сам часто делаю такие заявления о неясности языковых средств – по сути, в различных изданиях как раз этой книги.

Однако проблема в том, что подобные заявления в действительности применимы лишь к людям, работающим в одиночку и использующим исключительно код, который они написали самостоятельно. Как только *кто-то* в организации задействует "необязательное" расширенное языковое средство, оно перестает быть необязательным, а фактически навязывается *всем* в организации. То же самое относится к программному обеспечению, разработанному за пределами организации, которое вы применяете в своих системах – если его автор использовал сложное или чуждое языковое средство, тогда оно больше не будет необязательным для вас, поскольку вы должны понимать это средство, чтобы работать с кодом либо изменять его.

Следующее наблюдение касается всех *расширенных* тем, раскрытых в настоящей книге, в том числе перечисленных в качестве уровней "магии" в начале предыдущей главы и многих других. К ним относятся:

> *генераторы, декораторы, слоты, свойства, дескрипторы, метаклассы, диспетчеры контекстов, замыкания,* `super`*, пакеты пространств имен, Unicode, аннотации функций, относительное импортирование, аргументы с передачей только по ключевым словам, методы классов, статические методы и даже неясные сценарии применения включений и перегрузки операций.*

Если любой человек или программа, с которой вам необходимо работать, задействует такие инструменты, тогда они тоже становятся частью вашей *обязательной базы знаний.*

Чтобы посмотреть, насколько все способно быть обескураживающим, достаточно лишь принять во внимание процедуру *наследования нового стиля* из главы 40 – устрашающе запутанная модель, которая может сделать знание дескрипторов и метаклассов предварительным условием для понимания даже базового распознавания имен. Встроенная функция `super` из главы 32 аналогичным образом вносит свой вклад, навязывая чрезвычайно неявный и искусственный алгоритм MRO читателям любого кода, где используется `super`.

Совокупный эффект такого чрезмерного проектного решения приводит либо к радикальному усилению требований к обучению, либо к взращиванию базы пользователей, которые понимают применяемые ими инструменты только частично. Очевидно, что это далеко от идеала для тех, кто надеется использовать Python более простыми путями, и противоречит основной идее написания сценариев.

Против тревожных усовершенствований

Данное наблюдение также применимо ко многим *избыточным* свойствам, таким как метод `str.format` из главы 7 первого тома и оператор `with` из главы 34 — инструменты, которые позаимствованы из других языков и пересекаются с инструментами, давно присутствующими в Python. Когда программисты используют множество способов достижения той же самой цели, то все они становятся обязательным знанием.

Давайте будем честными: в последние годы Python изобилует избыточностью. Как я намекнул в предисловии, а вы удостоверились на личном опыте, современный мир Python переполнен дубликатами и расширениями функциональности, которые кратко описаны в табл. 41.1, помимо других упоминавшихся в книге.

Таблица 41.1. Выборочные примеры избыточности и взрывного роста средств в Python

Категория	Описание
3 главных парадигмы программирования	Процедурное, функциональное, объектно-ориентированное
2 несовместимых линейки	Python 2.X и Python 3.X с классами нового стиля в обеих
3 инструмента форматирования строк	Выражение `%`, `str.format`, `string.Template`
4 инструмента доступа к атрибутам	`__getattr__`, `__getattribute__`, свойства, дескрипторы
2 оператора финализации	`try/finally`, `with`
4 вида включений	Списковое включение, включение множества, включение словаря, генератор
3 инструмента дополнения классов	Вызовы функций, декораторы, метаклассы
4 вида методов	Методы экземпляра, статические методы, методы класса, методы метакласса
2 системы хранения атрибутов	Словари, слоты
4 разновидности импортирования	Импортирование модулей, импортирование пакетов, относительное импортирование пакетов, импортирование пакетов пространств имен
2 протокола координирования суперклассов	Прямые вызовы, `super` + MRO
5 форм оператора присваивания	Базовый, с множеством имен, дополненный, последовательности, со звездочкой
2 типа функций	Нормальная, генераторная
5 форм аргументов функций	Базовый, *имя=значение*, `*pargs`, `**kargs`, с передачей только по ключевым словам
2 источника поведения классов	Суперклассы, метаклассы
4 варианта предохранения состояния	Классы, замыкания, атрибуты функций, изменяемые объекты
2 модели классов	Классическая + нового стиля в Python 2.X, в Python 3.X классы нового стиля обязательны

Категория	Описание
2 модели Unicode	Необязательная в Python 2.X, обязательная в Python 3.X
2 режима PyDoc	Клиент с графическим пользовательским интерфейсом, требующий режима с единым браузером в последних версиях Python 3.X
2 схемы хранения байт-кода	Первоначальная, только `__pycache__` в последних версиях Python 3.X

Если вам небезразличен Python, тогда вы должны уделить время на просмотр табл. 41.1. Она отражает искусственно созданный взрывной рост функциональности и размера инструментального комплекта – 59 концепций, которые все вместе представляют собой непростое испытание для новоприбывших. Большинство категорий начиналось *только с одного первоначального элемента* в Python, многие были расширены отчасти для имитации других языков, и только несколько последних можно упростить, сделав вид, что последняя версия Python является единственной имеющей значение для программистов на этом языке.

Я подчеркивал, что в этой книге избегаю неоправданной сложности, но на практике как продвинутые, так и новые инструменты обычно подталкивают к их освоению – зачастую по причинам, связанным с непреодолимым желанием программиста продемонстрировать собственную удаль. Общим итогом оказывается то, что много кода на Python в наши дни изобилует такими сложными и чуждыми инструментами. То есть *ничто не является "необязательным" (в кавычках), если нет ничего по-настоящему необязательного.*

Сложность или мощь

Вот почему некоторых старожилов Python (включая меня) иногда беспокоит то, что Python с течением времени, похоже, становится все крупнее и сложнее. Новые средства, добавленные ветеранами, неофитами и даже любителями, могли слишком высоко поднять планку для новоприбывших. Хотя основные идеи языка Python вроде динамической типизации и встроенных типов по существу остались теми же, его расширенные дополнения могут стать обязательным чтением для любого программиста на Python. По этой причине я раскрываю здесь такие темы, невзирая на их отсутствие в предшествующих изданиях. Невозможно пропустить расширенные средства, если они есть в коде, который вам нужно понять.

С другой стороны, как упоминалось в главе 1 первого тома, для большинства наблюдателей язык Python все еще *заметно проще* многих своих современников и возможно сложен лишь настолько, насколько требуют его многочисленные роли. Несмотря на то что Python обзавелся многими такими же инструментами, как присутствующие в языках Java, C# и C++, они имеют тенденцию быть более легковесными в контексте динамически типизированного языка написания сценариев. При всем его росте с годами Python по-прежнему относительно легко изучать и использовать в сравнении с альтернативами, а изучающие его новички часто могут по мере необходимости выбирать сложные темы.

И, откровенно говоря, прикладные программисты, как правило, проводят большую часть своего времени, работая с *библиотеками и расширениями*, но не с продвину-

тыми и нередко загадочными языковыми средствами. Например, книга *Programming Python* (`http://www.oreilly.com/catalog/9780596158101`) в основном посвящена объединению Python с прикладными библиотеками, предназначенными для работы с графическими пользовательскими интерфейсами, базами данных и веб-сетью. Она не имеет дела с экзотическими языковыми инструментами (хотя Unicode все еще навязывает себя на многих этапах, и попутно неожиданно обнаруживается странное генераторное выражение и `yield`).

Кроме того, оборотной стороной такого роста является то, что Python стал *мощнее*. При надлежащем применении инструменты наподобие декораторов и метаклассов возможно не только "круты", но позволяют творческим программистам строить более гибкие и удобные API-интерфейсы для использования другими программистами. Как мы видели, они также способны предоставлять эффективные решения задач инкапсуляции и сопровождения.

Простота или элитарность

Оправдан ли потенциальный рост объема обязательных знаний Python – решать вам. Что бы там ни было, часто проблему решает уровень мастерства человека – более опытные программисты предпочитают более сложные инструменты и склонны забывать об их влиянии на другие стороны. К счастью, ситуация не является абсолютом; квалифицированные программисты также понимают, что *простота – это хорошая технология*, и сложные инструменты должны применяться только там, где они оправданы. Сказанное справедливо для любого языка программирования, но особенно для такого, как Python, который часто доступен начинающим программистам в качестве дополнительного инструмента.

Важно помнить о том, что многие люди, использующие Python, некомфортно себя чувствуют даже с *базовым объектно-ориентированным программированием*. Поверьте мне; я встречал тысячи таких. Хотя язык Python в принципе не считается тривиальной темой, рядовые разработчики программного обеспечения имеют совершенно четкое мнение: неоправданно добавленная сложность никогда не приветствуется, особенно если она обусловлена личными предпочтениями нерепрезентативного меньшинства. Независимо от того, было так задумано или нет, это часто вполне предсказуемо воспринимается как *элитарность* – непродуктивный и неприличный образ мыслей, которому не место в таком широко применяемом инструменте, как Python.

Конечно, проблема имеет социальный характер и касается как отдельных программистов, так и проектировщиков языка. Тем не менее, в "реальном мире", где программное обеспечение с открытым кодом подвергается оценке, основанные на Python системы, которые требуют от своих пользователей усвоения тонкостей метаклассов, дескрипторов и тому подобного, вероятно должны соответствующим образом отрегулировать ожидаемый объем потребления. Если эта книга выполнила свою работу, то вы поймете, что простота в программировании является одним из наиболее важных и долговременных решений.

Заключительные размышления

Итак, в вашем распоряжении ряд наблюдений от кое-кого, кто использовал, обучал и пропагандировал язык Python на протяжении более двух десятков лет и по-прежнему желает ему в будущем ничего кроме самого наилучшего. Разумеется, здесь нет ничего совершенно нового. В действительности рост объема самой книги, похоже, свидетельствует о влиянии собственного роста Python – если только не воспринимать

его как *ироничный панегирик* первоначальному представлению этого языка в качестве инструмента, который должен был упрощать программирование и быть доступным как экспертам, так и неспециалистам. Судя по одному лишь весу языка, по всей видимости, этой мечтой либо пренебрегли, либо вовсе от нее отказались.

Однако нынешний рост *популярности* Python не демонстрирует никаких признаков ослабления – сильный контраргумент в вопросах сложности. Сегодняшний мир Python по понятным причинам может быть менее озабочен достижением первоначальных и вероятно идеалистических целей, чем применением его имеющейся формы в своей работе. Язык Python многое делает в практическом мире сложных требований программирования, и это все еще достаточное основание рекомендовать его для решения многочисленных задач. Помимо исходных целей массовая притягательность квалифицируется как одна из форм успеха, хотя вердикт ее значимости должно вынести время.

Если вас интересуют дальнейшие размышления об эволюции и кривой обучения Python, тогда ознакомьтесь с моей статьей по ссылке `http://learning-python.com/pyquestions3.html`. Это важные практические вопросы, которые являются ключевыми для будущего Python и заслуживают большего внимания, чем я уделил им здесь. Но они крайне субъективны, а эта книга – не философский трактат, и вдобавок она уже успела превысить запланированный объем.

Более важно то, что в проекте с открытым кодом, подобном Python, ответы на такие вопросы должны вырабатываться заново каждой волной новоприбывших. Я надеюсь, что волна, с которой прибудете вы, принесет с собой столько же здравого смысла, сколько и веселья в выстраивании будущего Python.

Куда двигаться дальше?

Вот и все. Вы официально добрались до конца книги. Освоив Python вдоль и поперек, вашим следующим шагом, если только вы решитесь на него, будет исследование библиотек, методик и инструментов, доступных в предметных областях, где вам придется работать.

Поскольку язык Python настолько широко используется, вы обнаружите достаточно ресурсов для его применения практически в любом приложении, которое вы только можете себе представить – от графических пользовательских интерфейсов, веб-сети и баз данных до численного программирования, робототехники и системного администрирования. Ссылки на популярные инструменты и темы вы найдете в главе 1 первого тома и в веб-сети.

Именно здесь язык Python становится по-настоящему забавным, но на этом история настоящей книги заканчивается, и начинаются другие. Надеюсь вскоре вас увидеть в области программирования прикладных приложений.

Удачи вам в путешествии. А еще, конечно же, "всегда смотрите на светлую сторону жизни"!

На бис: распечатайте собственный сертификат об окончании!

И последнее: вместо упражнений в последней части книги я предлагаю вам бонусный сценарий для самостоятельного изучения и запуска. Я не могу предоставить сертификат об окончании каждому читателю книги (и даже если бы мог, то он все равно мало что стоит), но в состоянии включить непритязательный сценарий на Python,

который это сделает. В файле `certificate.py` с показанным далее содержимым находится сценарий на Python 2.X и 3.X, который создает простой сертификат об окончании чтения книги в форме текстового и HTML-файла, после чего отображает его в стандартном веб-браузере на вашем компьютере.

```
#!/usr/bin/python
"""
Файл certificate.py: сценарий на Python 2.X и 3.X.
Генерирует простой сертификат об окончании чтения: выводит
и сохраняет в текстовом и HTML-файле, отображаемом в веб-браузере.
"""
from __future__ import print_function  # Совместимость с Python 2.X
import time, sys, webbrowser

if sys.version_info[0] == 2:           # Совместимость с Python 2.X
    input = raw_input
    import cgi
    htmlescape = cgi.escape
else:
    import html
    htmlescape = html.escape

maxline  = 60                          # Для разделительных строк
browser  = True                        # Отображать в браузере
saveto   = 'Certificate.txt'           # Имена выходных файлов
template = """
%s

 ===> Official Certificate <===

Date: %s

This certifies that:

\t%s

has survived the massive tome:

\t%s

and is now entitled to all privileges thereof, including
the right to proceed on to learning how to develop Web
sites, desktop GUIs, scientific models, and assorted Apps,
with the possible assistance of follow-up applications
books such as Programming Python (shameless plug intended).

--Mark Lutz, Instructor

(Note: certificate void where obtained by skipping ahead.)

%s
"""

# Взаимодействие, настройка
for c in 'Congratulations!'.upper():
    print(c, end=' ')
    sys.stdout.flush()        # Иначе некоторые командные оболочки ждут \n
    time.sleep(0.25)
print()

date = time.asctime()
name = input('Enter your name: ').strip() or 'An unknown reader'
sept = '*' * maxline
book = 'Learning Python 5th Edition'
```

```
# Создание версии в текстовом файле
file = open(saveto, 'w')
text = template % (sept, date, name, book, sept)
print(text, file=file)
file.close()

# Создание версии в файле html
htmlto = saveto.replace('.txt', '.html')
file = open(htmlto, 'w')

tags = text.replace(sept,  '<hr>')     # Вставка нескольких html-дескрипторов
tags = tags.replace('===>', '<h1 align=center>')
tags = tags.replace('<===', '</h1>')

tags = tags.split('\n')                 # Построчный режим
tags = ['<p>' if line == ''
        else line for line in tags]
tags = ['<i>%s</i>' % htmlescape(line) if line[:1] == '\t'
        else line for line in tags]
tags = '\n'.join(tags)

link = '<i><a href="http://www.rmi.net/~lutz">Book support site</a></i>\n'
foot = '<table>\n<td><img src="ora-lp.jpg" hspace=5>\n<td>%s</table>\n' % link
tags = '<html><body bgcolor=beige>' + tags + foot + '</body></html>'

print(tags, file=file)
file.close()

# Отображение результатов
print('[File: %s]' % saveto, end='')
print('\n' * 2, open(saveto).read())

if browser:
   webbrowser.open(saveto, new=True)
   webbrowser.open(htmlto, new=False)

if sys.platform.startswith('win'):
   input('[Press Enter]')    # Оставить окно открытым при щелчке в Windows
```

Запустите сценарий `certificate.py` самостоятельно и изучите его код, чтобы подытожить ряд идей, которые были раскрыты в книге. Он доступен в пакете кода примеров. В его коде вы не найдете дескрипторов, декораторов, метаклассов или вызовов `super`, но все же это типичный сценарий на Python.

В результате запуска сценарий сгенерирует веб-страницу, представленную на рис. 41.1. Конечно, все могло бы выглядеть гораздо более пышно, т.к. Python поддерживает инструменты для работы с документами PDF и другими форматами, например, Sphinx из главы 15 первого тома. Но ведь вы в конце книги, а потому заслужили еще одну шутку или две...

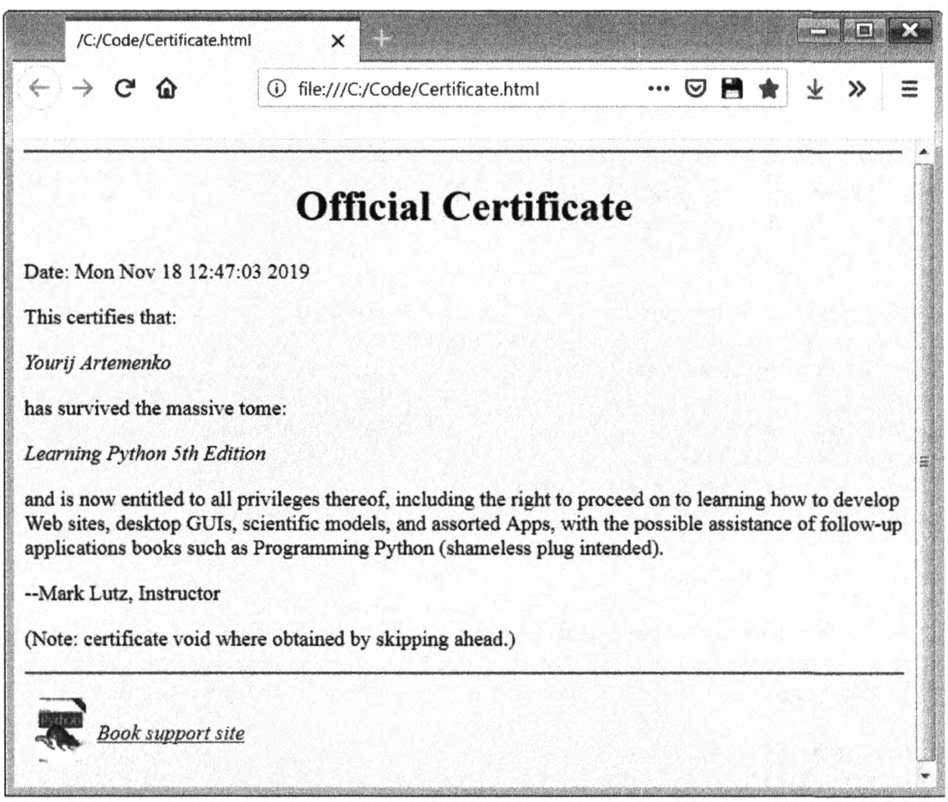


Рис. 41.1. *Веб-страница, созданная и открытая сценарием* `certificate.py`

ЧАСТЬ IX

Приложения

Приложение А

Установка и конфигурирование

В этом приложении представлены дополнительные сведения об установке и конфигурировании, предназначенные для тех, кому подобные темы в новинку. Они вынесены сюда потому, что далеко не всем читателям придется иметь дело с ними дело непосредственно. Однако из-за того, что здесь рассматриваются сопутствующие темы вроде переменных среды и аргументов командной строки, материал данного приложения заслуживает хотя бы беглого ознакомления.

Установка интерпретатора Python

Поскольку для запуска сценариев на Python необходим интерпретатор Python, первым шагом в использовании языка обычно является установка интерпретатора Python. Если он еще не доступен на вашем компьютере, то вам придется получить, установить и возможно сконфигурировать последнюю версию Python. Делать это нужно только один раз на каждом компьютере, а если вы будете запускать фиксированный двоичный файл (описанный в главе 2 первого тома) или самоустанавливающуюся систему, тогда шаги установки могут быть тривиальными или вообще отсутствовать.

Присутствует ли Python на компьютере?

Прежде чем делать что-либо еще, проверьте, имеется ли на компьютере последняя версия Python. Если вы работаете в среде Linux, Mac OS X или Unix, то интерпретатор Python вероятно уже установлен, хотя его версия может быть не самой новой. Ниже описаны необходимые действия.

- В *Windows 7* и предшествующих версиях проверьте, существует ли группа Python в меню Все программы, доступном через кнопку Пуск. В *Windows 8* и последующих версиях поищите Python в плитках на начальном экране, в интерфейсе All apps (Все приложения) на экране Start (Пуск) или в проводнике файлов.
- В *Mac OS X* откройте окно терминала (Applications⇨Utilities⇨Terminal (Приложения⇨Служебные⇨Терминал)) и введите `python` в командной строке. В данной системе интерпретатор Python, среда IDLE и комплект инструментов `tkinter` являются стандартными компонентами.

- В *Linux* и *Unix* введите python в приглашении командной оболочки (т.е. в окне терминала) и посмотрите, что произойдет. В качестве альтернативы попробуйте поискать "python" в обычных местах – /usr/bin, /usr/local/bin и т.д. Подобно Mac OS X в системах Linux интерпретатор Python является стандартным компонентом.

Если вы обнаружили интерпретатор Python, тогда удостоверьтесь в том, что он последней версии. Хотя в большей части книги подойдет любой из последних выпусков Python, основное внимание здесь сосредоточено на версиях Python 3.3 (3.7) и 2.7, поэтому имеет смысл установить именно указанные версии.

Говоря о *версиях*, рекомендуется начать с Python 3.3 или последующей версии, если вы изучаете Python с нуля и не нуждаетесь в работе с существующим кодом Python 2.X; в противном случае вы должны применять Python 2.7. Тем не менее, ряд популярных систем, основанных на Python, по-прежнему используют старые выпуски (Python 2.6 и даже Python 2.5 все еще широко распространены), а потому если вы имеете дело с существующими системами, то выбирайте версию в соответствии со своими потребностями. В следующем разделе описаны места, где можно получить разнообразные версии Python.

Где взять интерпретатор Python

Если интерпретатор Python на вашем компьютере отсутствует, тогда вам придется установить его самостоятельно. Хорошая новость в том, что Python является системой с открытым кодом, которая свободно доступна в веб-сети и очень легко устанавливается на большинстве платформ.

Вы всегда можете получить самый последний выпуск *стандартного Python* на официальном веб-сайте Python по адресу http://www.python.org. Найдите на странице ссылку Downloads (Загрузки) и выберите выпуск для платформы, где вы будете работать. Вы обнаружите установщики для *Windows* и *Mac OS X*, полный исходный код (обычно требующий компиляции на компьютере с Linux, Unix или OS X для получения интерпретатора) и другие ресурсы.

Несмотря на то что в настоящее время Python является стандартным компонентом в *Linux*, вы можете также найти пакеты RPM для Linux в веб-сети (распаковывайте их с помощью утилиты rpm). На официальном веб-сайте Python также есть ссылки на страницы, где поддерживаются версии для других платформ, либо на самом веб-сайте python.org, либо за его пределами. Например, вы обнаружите сторонние установщики Python для Google *Android*, а также приложения для установки Python в среде Apple *iOS*.

Поиск с помощью Google – еще один великолепный способ найти пакеты установки Python. Среди других платформ вы заметите предварительно собранный интерпретатор Python для iPod, карманных устройств Palm, смартфонов Nokia, игровых консолей PlayStation и PSP, Solaris, AS/400 и Windows Mobile, хотя некоторые из них обычно отстают на несколько выпусков.

Если вы предпочитаете иметь среду Unix на компьютере с Windows, тогда вас может также заинтересовать установка средства *Cygwin* и его версии Python (http://www.cygwin.com). Cygwin – это подпадающие под действие лицензии GPL библиотека и инструментальный комплект, предоставляющие полную функциональность Unix на компьютерах Windows и включающие предварительно собранный интерпретатор Python, который задействует все инструменты Unix.

Вы также обнаружите Python на компакт-дисках с дистрибутивами Linux. Как правило, версии будут немного отставать от текущей, но обычно незначительно.

Кроме того, вы можете найти Python в рамках ряда бесплатных и коммерческих пакетов для разработки. Ниже перечислены некоторые альтернативные дистрибутивы.

ActiveState ActivePython

Пакет, который комбинирует Python с расширениями для научных расчетов, операционной системы Windows и других потребностей разработки, включая PyWin32 и IDE-среду PythonWin.

Enthought Python Distribution

Комбинация Python с множеством дополнительных библиотек и инструментов, ориентированных на научные расчеты.

Portable Python

Смесь Python и добавочных пакетов, сконфигурированная для запуска непосредственно на портативном устройстве.

Pythonxy

Дистрибутив Python на основе Qt и Spyder, ориентированный на научные расчеты.

Conceptive Python SDK

Пакет, нацеленный на производственные, настольные и приложения для работы с базами данных.

PyIMSL Studio

Коммерческий дистрибутив для численного анализа.

Anaconda Python

Дистрибутив для анализа и визуализации крупных наборов данных.

Приведенный перечень подвержен изменениям, поэтому ищите в веб-сети актуальную информацию по всем перечисленным и другим дистрибутивам. Одни дистрибутивы бесплатны, другие нет, а некоторые имеют как бесплатные, так и платные версии. Все они объединяют стандартный Python, бесплатно доступный на веб-сайте `http://www.python.org`, с дополнительными инструментами, но способны упростить установку для многих пользователей.

Наконец, если вас интересуют альтернативные реализации Python, то поищите в веб-сети сведения о *Jython* (перенесенная версия Python для среды Java) и *IronPython* (Python для мира C#/.NET), которые были кратко описаны в главе 2 первого тома. Рассмотрение установки упомянутых систем выходит за рамки книги.

Шаги установки

После того, как вы загрузили интерпретатор Python, его понадобится установить. Шаги установки крайне специфичны к платформе, но ниже дается несколько подсказок для основных платформ, поддерживаемых Python (с перекосом в сторону Windows, т.к. вероятно именно здесь начинают знакомиться с Python большинство новоприбывших).

Windows

Для Windows интерпретатор Python поставляется как *самоустанавливающийся* файл MSI – просто дважды щелкните на его значке и щелкайте на кнопках Yes (Да) или Next (Далее) в открываемых впоследствии окнах, чтобы выполнить стандартную установку. Стандартная установка включает набор документации по Python, а также поддержку для `tkinter` (`Tkinter` в Python 2.X), баз данных `shelve` и графического пользовательского интерфейса разработки IDLE. Версии Python 3.7 и 2.7 обычно устанавливаются в каталоги `C:\Python37` и `C:\Python27`, хотя во время установки их можно изменить.

Ради удобства в Windows 7 и предшествующих версиях Python появляется в меню Все программы кнопки Пуск. Меню Python содержит пять элементов, которые обеспечивают быстрый доступ к распространенным задачам: запуск пользовательского интерфейса IDLE, чтение документации по модулям, запуск интерактивного сеанса, чтение стандартных руководств по Python и удаление. С большинством элементов связаны концепции, детально исследованные в других местах книги.

Во время установки в Windows интерпретатор Python также автоматически использует *файловые ассоциации*, чтобы зарегистрировать себя как программу, которая открывает файлы Python при щелчке на их значках (методика запуска программ была описана в главе 3 первого тома). В Windows можно также скомпилировать интерпретатор Python из его исходного кода, но обычно это не делается, и потому мы не будем рассматривать здесь все детали (см. `python.org`).

Есть еще одно замечание относительно установки в Windows. В следующем приложении приводится введение в *запускающий модуль Windows*, который поставляется, начиная с Python 3.3. Он изменяет ряд правил для установки, файловых ассоциаций и командных строк, но может служить ценным ресурсом, если вы имеете на своем компьютере несколько версий Python (скажем, Python 2.X и Python 3.X). Согласно приложению Б установщик MSI для Python 3.X также предлагает возможность установки переменной среды `PATH`, чтобы включить каталог с установленным интерпретатором Python.

Linux

В среде Linux при отсутствии интерпретатора Python вы, скорее всего, можете получить его в форме одного или большего числа файлов RPM и распаковать их обычным образом. В зависимости от того, какие файлы RPM вы загрузили, один из них может быть предназначен для самого Python, а еще один – для добавления поддержки `tkinter` и среды IDLE. Поскольку операционная система Linux похожа на Unix, следующий абзац применим также и к ней.

Unix

Для систем Unix интерпретатор Python обычно компилируется из полного исходного кода на C. Как правило, это требует лишь распаковки файла и запуска команд `configure` и `make`; процедура построения интерпретатора Python конфигурируется автоматически в соответствии с системой, где он компилируется. Однако обязательно почитайте файл `README`, чтобы ознакомиться с дополнительными сведениями о процессе сборки. Так как интерпретатор Python относится к системам с открытым кодом, его исходный код можно свободно использовать и распространять.

На других платформах детали установки могут варьироваться в широких пределах, но в целом следуют нормальным соглашениям, принятым внутри платформы. Например, установка перенесенной версии Python для PalmOS под названием Pippy требует синхронизации с вашим карманным устройством, а Python для карманного устройства Sharp Zaurus, основанного на Linux, выглядел как один или несколько файлов `.ipk`, которые просто запускались с целью установки (вероятно, такие устройства все еще функционируют, но отыскать их в наши дни может быть нелегко).

Кроме того, интерпретатор Python можно устанавливать и применять на платформах *Android* и *iOS*, но методики установки и использования слишком специфичны к платформам, чтобы раскрывать их здесь. Описания дополнительных процедур установки и свежие новости о доступных перенесенных версиях ищите на веб-сайте Python.

Конфигурирование интерпретатора Python

После установки интерпретатора Python у вас может возникнуть желание сконфигурировать ряд системных настроек, которые оказывают влияние на то, каким образом интерпретатор Python запускает ваш код. (Если вы только приступили к изучению языка, тогда можете пропустить весь текущий раздел; для базовых программ обычно нет необходимости устанавливать какие-либо системные настройки.)

В общем случае линии поведения интерпретатора Python можно конфигурировать с помощью настройки переменных среды и параметров командной строки. В настоящем разделе мы кратко обсудим оба варианта, но обязательно ознакомьтесь с дополнительными сведениями по представленным здесь темам, обратившись в документацию.

Переменные среды Python

Переменные среды, также известные как переменные командной оболочки или переменные DOS, являются настройками уровня системы, которые находятся вне интерпретатора Python и потому могут применяться для настройки поведения интерпретатора каждый раз, когда он запускается на отдельно взятом компьютере. Интерпретатор Python распознает несколько переменных среды, но лишь некоторые из них используются достаточно часто, чтобы заслуживать подробного объяснения. В табл. А.1 приведена сводка по основным переменным среды, касающимся Python (информацию об остальных ищите на справочных ресурсах Python).

Таблица А.1. Важные переменные среды

Переменная	Роль
`PATH` (или `path`)	Путь поиска командной оболочки системы (для нахождения `python`)
`PYTHONPATH`	Путь поиска модулей Python (для операций импортирования)
`PYTHONSTARTUP`	Путь к файлу запуска для интерактивной оболочки Python
`TCL_LIBRARY`, `TK_LIBRARY`	Переменные для расширения `tkinter`
`PY_PYTHON`, `PY_PYTHON3`, `PY_PYTHON2`	Стандартные настройки для запускающего модуля Windows (см. приложение Б)

Хотя использовать перечисленные переменные легко, ниже дается ряд подсказок.

PATH

Настройка PATH содержит список каталогов, в которых операционная система ищет исполняемые программы, когда они запускаются без указания полного пути к каталогу. Обычно она должна включать каталог, где находится интерпретатор Python (программа python в Unix, файл python.exe в Windows).

Вам вообще не нужно устанавливать эту переменную, если вы собираетесь работать в каталоге, в котором расположен интерпретатор Python, или вводить полный путь к Python в командной строке. Скажем, в Windows переменная среды PATH не играет никакой роли, если перед запуском любого кода вы вводите команду cd C:\Python37 (чтобы перейти в каталог, где находится Python — хотя, как объяснялось в главе 3 первого тома, хранить там собственный код не рекомендуется) или всегда набираете C:\Python37\python (с полным путем) вместо просто python.

Также обратите внимание, что настройка PATH предназначена главным образом для запуска программ из командной строки; она не имеет какого-либо значения при запуске через щелчки на значках и IDE-среду — в первом случае применяются файловые ассоциации, а во втором используются встроенные механизмы, поэтому такой шаг конфигурирования не требуется. В приложении Б описан вариант автоматической настройки PATH во время установки.

PYTHONPATH

Роль настройки PYTHONPATH похожа на роль PATH: интерпретатор Python обращается к переменной PYTHONPATH при определении местонахождения файлов модулей, когда вы *импортируете* их в программе. В переменной PYTHONPATH указывается зависимый от платформы список имен каталогов, разделяемых двоеточиями в Unix и точками с запятой в Windows. Список обычно включает только каталоги с вашим исходным кодом. Содержимое PYTHONPATH объединяется в путь поиска импортируемых модулей sys.path вместе с контейнерным каталогом сценария, любыми настройками файлов путей .pth и каталогами стандартных библиотек.

Вам не понадобится устанавливать переменную PYTHONPATH, если вы не планируете выполнять *импортирование между каталогами*. Поскольку интерпретатор Python всегда автоматически производит поиск файла верхнего уровня в домашнем каталоге программы, данная настройка требуется, только если модулю необходимо импортировать еще один модуль, который находится в другом каталоге. Ознакомьтесь также с обсуждением файлов путей .pth позже в текущем приложении для создания альтернативы PYTHONPATH. Дополнительные сведения о пути поиска модулей ищите в главе 22 первого тома.

PYTHONSTARTUP

В случае установки переменной среды PYTHONSTARTUP в путь к файлу с кодом на Python интерпретатор Python автоматически выполняет код из файла всякий раз, когда вы запускаете интерактивный сеанс, как если бы он вводится в командной строке интерактивного сеанса. Переменная PYTHONSTARTUP применяется редко, но является удобным способом гарантировать, что вы всегда загружаете определенные утилиты при работе в интерактивном сеансе; она сохраняет импортирование каждый раз, когда вы запускаете сеанс Python.

Настройки `tkinter`

Если вы хотите использовать инструментальный комплект для построения графических пользовательских интерфейсов под названием `tkinter` (`Tkinter` в Python 2.X), то можете указать в переменных `TCL_LIBRARY` и `TK_LIBRARY` имена каталогов библиотек систем Tcl и Tk (почти как `PYTHONPATH`). Тем не менее, в системах Windows подобные настройки не обязательны (поддержка `tkinter` устанавливается параллельно с Python), а также обычно не требуются в системах Mac OS X и Linux, если только базовые библиотеки Tcl и Tk не стали недействительными или не находятся в нестандартных каталогах (дополнительные детали ищите на странице Download веб-сайта `python.org`).

PY_PYTHON, PY_PYTHON3, PY_PYTHON2

Настройки `PY_PYTHON`, `PY_PYTHON3` и `PY_PYTHON2` применяются для указания стандартных версий Python, когда вы используете запускающий модуль Windows, который поставляется, начиная с версии Python 3.3. Запускающий модуль Windows подробно рассматривается в приложении Б.

Так как описанные выше переменные среды являются внешними по отношению к самому интерпретатору Python, *момент* их модификации неважен: изменения можно вносить до или после установки Python, но перед действительным *запуском* интерпретатора Python – чтобы изменения вступили в силу, обязательно перезапустите IDE-среду и интерактивный сеанс Python.

`tkinter` и IDLE в Linux и Mac OS

Среда IDLE, описанная в главе 3 первого тома, представляет собой программу на Python с графическим пользовательским интерфейсом `tkinter`. Модуль `tkinter` (`Tkinter` в Python 2.X) – это инструментальный комплект для построения графических пользовательских интерфейсов, который автоматически устанавливается вместе со стандартным Python в Windows и является неотъемлемой частью систем Mac OS X и Linux.

Однако в некоторых системах *Linux* внутренняя библиотека для построения графических пользовательских интерфейсов может не быть стандартным устанавливаемым компонентом. Чтобы при необходимости добавить такую поддержку к Python в Linux, попробуйте ввести командную строку вида `yum install tkinter` для автоматической установки библиотек, лежащих в основе `tkinter`. Прием должен работать в дистрибутивах Linux (и ряде других систем), где доступна программа установки `yum`; в противном случае ищите сведения в документации по установке для имеющейся платформы.

Как также обсуждалось в главе 3 первого тома, среда IDLE в *Mac OS X* вероятно находится в папке `MacPython` (или `Python N.M`) внутри папки `Applications` (наряду с `PythonLauncher`, используемым для запуска программ с помощью щелчков в Finder), но обязательно просмотрите страницу Downloads на веб-сайте `python.org`, если со средой IDLE возникают трудности; в некоторых версиях Mac OS X может потребоваться установить обновление.

Способы установки конфигурационных параметров

Способ установки переменных среды, относящихся к Python, и то, во что их устанавливать, зависит от типа компьютера, на котором вы работаете. И снова помните

о том, что вы не обязаны устанавливать их немедленно; в случае, если вы работаете в IDLE (см. главу 3 первого тома) и храните все свои файлы в том же самом каталоге, заблаговременное конфигурирование вероятно не потребуется.

Но предположим, например, что у вас есть в целом полезные файлы модулей в каталогах `utilities` и `package1` где-то на компьютере, и вы хотите иметь возможность импортировать указанные модули в файлах, расположенных в других каталогах. То есть для загрузки файла по имени `spam.py` из каталога `utilities` или `package1` вы желаете указывать в другом файле оператор следующего вида:

```
import spam
```

Чтобы заставить это работать, вам тем или иным образом придется сконфигурировать путь поиска модулей для включения каталога, содержащего `spam.py`. Ниже приводится несколько советов относительно такого процесса с применением `PYTHONPATH` в качестве примера; при необходимости проделайте то же самое с другими настройками вроде `PATH` (начиная с Python 3.3, переменная среды `PATH` может устанавливаться автоматически: см. приложение Б).

Переменные командной оболочки Unix/Linux

В системах Unix способ установки переменных среды зависит от используемой командной оболочки. В командной оболочке `csh` вы можете добавить в файл `.cshrc` или `.login` показанную ниже строку, чтобы установить путь поиска модулей Python:

```
setenv PYTHONPATH /usr/home/pycode/utilities:/usr/lib/pycode/package1
```

Это сообщает интерпретатору Python о том, что импортируемые модули нужно искать в двух пользовательских каталогах. Если же вы применяете командную оболочку `ksh`, тогда установка будет располагаться в файле `.kshrc` и выглядеть следующим образом:

```
export PYTHONPATH="/usr/home/pycode/utilities:/usr/lib/pycode/package1"
```

В других командных оболочках может использоваться отличающийся (но аналогичный) синтаксис.

Переменные DOS (и более старые версии Windows)

Если вы работаете в MS-DOS или какой-то теперь уже довольно старой версии Windows, то вам понадобится добавить команду конфигурирования переменной среды в файл `C:\autoexec.bat` и перезагрузить систему, чтобы изменения вступили в силу. Команда конфигурирования на таких компьютерах имеет синтаксис, уникальный для DOS:

```
set PYTHONPATH=c:\pycode\utilities;d:\pycode\package1
```

Вы можете ввести такую команду также в окне консоли DOS, но тогда настройка будет активной лишь для этого одного окна консоли. Помещение изменения в файл `.bat` делает его постоянным и глобальным для всех программ, хотя в последние годы такая методика была замещена так, как описано в следующем разделе.

Графический пользовательский интерфейс для установки переменных среды Windows

Во всех последних версиях Windows вы можете устанавливать `PYTHONPATH` и другие переменные через графический пользовательский интерфейс для установки переменных среды, не занимаясь редактированием файлов, вводом командных строк

и перезагрузкой системы. Откройте панель управления, выберите элемент Система, щелкните на вкладке или ссылке Дополнительные параметры системы и затем на кнопке Переменные среды, чтобы редактировать или добавлять новые переменные (PYTHONPATH обычно будет новой пользовательской переменной). Применяйте такое же имя и значение переменной, как в команде set в предыдущем разделе.

Перезагрузка системы не требуется, но обязательно перезапустите интерпретатор Python, если он функционировал, чтобы изменения вступили в силу – путь поиска импортируемых модулей настраивается только во время начального запуска. В случае работы в окне командной строки Windows также возможно понадобится его перезапуск для подхвата изменений.

Реестр Windows

Если вы являетесь опытным пользователем Windows, то можете также сконфигурировать путь поиска модулей, используя редактор реестра Windows. Чтобы открыть редактор реестра, введите regedit в окне, появляющемся в результате выбора пункта меню Пуск⇨Выполнить..., в поле поиска ниже кнопки Пуск или в окне командной строки. При условии, что редактор реестра доступен в вашей системе, вы можете перейти к записям Python и внести желаемые изменения. Тем не менее, процедура довольно тонкая и подвержена ошибкам, так что если вы не очень хорошо знакомы с реестром, тогда рекомендуется избрать другие варианты (на самом деле это похоже на хирургию головного мозга для вашего компьютера, а потому будьте предельно осторожны!).

Файлы путей

Наконец, если вы предпочитаете расширять путь поиска модулей с помощью файла путей .pth, а не переменной PYTHONPATH, то можете создать текстовый файл (скажем, C:\Python37\mypath.pth), содержимое которого в Windows выглядит следующим образом:

```
c:\pycode\utilities
d:\pycode\package1
```

Его содержимое будет варьироваться в зависимости от платформы, а контейнерный каталог может отличаться для разных платформ и выпусков Python. Интерпретатор Python автоматически находит данный файл во время своего начального запуска.

Имена каталогов в файлах путей могут быть абсолютными или относительными к каталогу, где эти файлы хранятся; допускается применять множество файлов .pth (добавятся все указанные в них каталоги), и файлы .pth могут присутствовать в разнообразных автоматически проверяемых каталогах, которые являются специфичными к платформе и версии. В целом выпуск Python, пронумерованный как Python *N.M*, обычно ищет файлы путей внутри каталогов C:\PythonNM и C:\PythonNM\Lib\site-packages в Windows и внутри каталогов /usr/local/lib/pythonN.M/site-packages и /usr/local/lib/site-python в Unix/Linux. Использование файлов путей для конфигурирования пути поиска импортируемых модулей sys.path более подробно описано в главе 22 первого тома.

Поскольку настройки среды часто необязательны, а книга не посвящена командным оболочкам операционных систем, за дополнительными деталями обращайтесь к другим источникам. Просмотрите страницы руководства по своей командной оболочке или другую документацию, а при возникновении трудностей с выяснением, как должны выглядеть ваши настройки, проконсультируйтесь с системным администратором или другим местным экспертом.

Аргументы командной строки интерпретатора Python

При запуске интерпретатора Python из командной строки системы (т.е. приглашения командной оболочки или окна командной строки) вы можете передавать разнообразные аргументы для управления тем, как он будет выполнять ваш код. В отличие от переменных среды уровня системы, которые были описаны в предыдущем разделе, аргументы командной строки могут быть разными каждый раз, когда вы запускаете свой сценарий. Полная форма командной строки для вызова интерпретатора Python 3.7 выглядит примерно так (в Python 2.7 она почти такая же, но с небольшими отличиями):

```
python [-bBdEhiIOqsSuvVWxX] [-c команда | -m имя-модуля | сценарий | - ] [аргументы]
```

Ниже будут кратко представлены наиболее часто применяемые аргументы интерпретатора Python. Дополнительные детали о доступных аргументах командной строки, которые здесь не раскрываются, ищите в руководствах или справочниках по Python. А еще лучше запросите у самого интерпретатора Python с помощью такой команды:

```
C:\code> python -h
```

Интерпретатор Python отобразит справочный текст с описанием всех доступных аргументов командной строки. Если вы имеете дело со сложными командными строками, тогда обязательно ознакомьтесь со стандартными библиотечными модулями в данной области: первоначальным `getop`, более новым `argparse` и теперь устаревшим (начиная с Python 3.2) `optparse`, которые поддерживают развитую обработку командных строк. Также просмотрите руководства и справочники по стандартной библиотеке Python, чтобы узнать больше о задействованных далее модулях `pdb` и `profile`.

Запуск файлов сценариев с аргументами

В большинстве случаев из формата командной строки Python используются только части *сценарий* и *аргументы* для запуска файла с исходным кодом программы и передачи аргументов, потребляемых самой программой. В целях иллюстрации рассмотрим показанный ниже сценарий, находящийся в текстовом файле по имени `showargs.py` внутри `C:\code` или другого каталога по вашему выбору. Он выводит аргументы командной строки, доступные сценарию как `sys.argv`, который представляет собой список Python строк Python (способы создания и запуска файлов сценариев Python обсуждались в главах 2 и 3 первого тома; здесь нас интересуют лишь аргументы командной строки):

```
# Файл showargs.py
import sys
print(sys.argv)
```

В следующей команде части `python` и `showargs.py` могут также включать полные пути к каталогам — первая полагается на переменную среды `PATH`, а вторая на то, что файл сценария находится в текущем каталоге. Три аргумента (`a b -c`), предназначенные для сценария, обнаруживаются в списке `sys.argv` и могут там инспектироваться кодом сценария; первым элементом в `sys.argv` всегда будет имя файла сценария, когда оно известно:

```
C:\code> python showargs.py a b -c      # Самая распространенная команда:
                                        #   запуск файла сценария
['showargs.py', 'a', 'b', '-c']
```

Как объяснялось в разных местах книги, *списки* Python выводятся в квадратных скобках, а *строки* отображаются в кавычках.

Запуск кода, предоставляемого в аргументах и стандартном входном потоке

Другие варианты спецификации формата кода позволяют предоставлять интерпретатору Python код, подлежащий запуску, в самой командной строке (-c) и принимать код из стандартного входного потока (символ — означает чтение из канала или входного потока, перенаправленного в файл):

```
C:\code> python -c "print(2 ** 100)"    # Читать код из аргумента
                                        #   командной строки
1267650600228229401496703205376

C:\code> python -c "import showargs" # Импортировать файл для запуска его кода
['-c']

C:\code> python - < showargs.py a b -c  # Читать код из стандартного
                                        #   входного потока
['-', 'a', 'b', '-c']

C:\code> python - a b -c < showargs.py  # Тот же результат, что и в
                                        #   предыдущей команде
['-', 'a', 'b', '-c']
```

Запуск модулей в пути поиска

Спецификация кода -m обеспечивает нахождение модуля в пути поиска модулей Python и его запуск в качестве сценария верхнего уровня (как модуля `__main__`). То есть интерпретатор Python ищет сценарий тем же самым способом, что и операции импортирования, с применением списка каталогов, обычно известного как sys.path, который включает текущий каталог, настройки PYTHONPATH и стандартные библиотеки. Расширение .py здесь не указано, потому что имя файла трактуется как модуль.

```
C:\code> python -m showargs a b -c    # Отыскать/запустить модуль как сценарий
['c:\\code\\showargs.py', 'a', 'b', '-c']
```

Аргумент -m также поддерживает запуск инструментов, модулей в пакетах с относительным импортированием и без него и модулей, находящихся в архивах .zip. Скажем, -m часто используется для запуска модулей отладчика pdb и профилировщика profile из командной строки, чтобы инициировать работу сценария вместо его выполнения в интерактивном режиме:

```
C:\code> python                         # Интерактивный сеанс отладчика
>>> import pdb
>>> pdb.run('import showargs')
...остальное не показано: см. документацию по pdb...

C:\code> python -m pdb showargs.py a b -c  # Отладка сценария (c=продолжить)
> C:\code\showargs.py(2)<module>()
-> import sys
(Pdb) c
['showargs.py', 'a', 'b', '-c']
...остальное не показано: q для выхода...
```

Профилировщик запускает и измеряет время выполнения вашего кода; его вывод может варьироваться в зависимости от выпуска Python, операционной системы и компьютера:

```
C:\code> python -m profile showargs.py a b -c    # Профилирование сценария
['showargs.py', 'a', 'b', '-c']
         9 function calls in 0.016 seconds

   Ordered by: standard name
```

```
ncalls tottime percall cumtime percall filename:lineno(function)
     2 0.000 0.000 0.000 0.000 :0(charmap_encode)
     1 0.000 0.000 0.000 0.000 :0(exec)
...остальное не показано: см. документацию по profile...
```

Вы также можете применять аргумент -m, чтобы породить процесс для программы с графическим пользовательским интерфейсом IDLE (см. главу 3 первого тома), находящейся в стандартной библиотеке, из другого каталога. Кроме того, посредством аргумента -m вы можете запускать модули инструментов pydoc и timeit с командными строками, как делалось в главах 15 и 21 первого тома:

```
c:\code> python -m idlelib.idle -n # Запустить IDLE в пакете, без подпроцессов
c:\code> python -m pydoc -b    # Запустить модули инструментов pydoc и timeit
c:\code> python -m timeit -n 1000 -r 3 -s "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
```

Оптимизированный и небуферизированный режимы

Непосредственно после слова python и перед указанием кода, подлежащего выполнению, интерпретатор Python принимает дополнительные аргументы, которые управляют его собственным поведением. Эти аргументы потребляются самим интерпретатором Python и не предназначены для выполняемого сценария. Например, -O запускает Python в оптимизированном режиме, а -u вынуждает стандартные потоки быть небуферизированными — в последнем случае любой выводимый текст будет немедленно финализирован, не задерживаясь в буфере:

```
C:\code> python -O showargs.py a b -c    # Оптимизированный режим: построить/
                                         #   запустить байт-код .pyo
C:\code> python -u showargs.py a b -c    # Небуферизированный стандартный
                                         #   выходной поток
```

Интерактивный режим после выполнения

В заключение флаг -i обеспечивает вход в интерактивный режим после выполнения сценария, что особенно полезно в качестве инструмента отладки, поскольку вы можете выводить финальные значения переменных после успешного запуска для получения дополнительных деталей:

```
C:\code> python -i showargs.py a b -c   # Перейти в интерактивный режим после
                                        #   завершения сценария
['showargs.py', 'a', 'b', '-c']
>>> sys                    # Финальное значение sys: импортированный модуль
<module 'sys' (built-in)>
>>> ^Z
```

Вдобавок вы можете выводить таким способом переменные после того, как исключение привело к прекращению работы сценария, чтобы посмотреть, каким образом они выглядели, когда произошло исключение, даже если сценарий не был запущен в режиме отладки — хотя здесь также можно запустить инструмент анализа отладчика (type является командой отображения содержимого файлов в Windows; в других системах попробуйте cat или что-нибудь еще):

```
C:\code> type divbad.py
X = 0
print(1 / X)

C:\code> python divbad.py          # Запустить сценарий, содержащий ошибку
...текст сообщения об ошибке не показан...
```

```
ZeroDivisionError: division by zero
Ошибка деления на ноль: деление на ноль

C:\code> python -i divbad.py        # Вывести значения переменных после
                                     #   возникновения ошибки
...текст сообщения об ошибке не показан...
ZeroDivisionError: division by zero
Ошибка деления на ноль: деление на ноль

>>> X
0
>>> import pdb                       # Теперь запустить полный сеанс отладки
>>> pdb.pm()
> C:\code\divbad.py(2)<module>()
-> print(1 / X)
(Pdb) quit
```

Аргументы командной строки Python 2.X

Помимо упомянутых выше интерпретатор Python 2.7 поддерживает аргументы, которые содействуют совместимости с Python 3.X (`-3` для выдачи предупреждений о несовместимостях и `-Q` для управления моделями операции деления) и обнаруживают несогласованное использование отступов посредством табуляции, что всегда делается в Python 3.X (`-t`; см. главу 12). При необходимости вы всегда можете запросить информацию о доступных аргументах у самого интерпретатора Python 2.X:

```
C:\code> c:\python27\python -h
```

Командные строки запускающего модуля, появившегося в Python 3.3

Формально в предыдущем разделе были описаны аргументы, которые вы можете передавать самому интерпретатору Python — программе, обычно называющейся `python.exe` в Windows и `python` в Linux (в Windows расширение `.exe`, как правило, опускается). В приложении Б вы увидите, что запускающий модуль, появившийся в Python 3.3, дополняет эту историю для пользователей Python 3.3 и последующих версий или автономного пакета запуска. Он добавляет новые исполняемые файлы, которые принимают номера версий как аргументы в командных строках, применяемых для запуска интерпретатора Python и ваших сценариев (файл `what.py` будет представлен в приложении Б; он просто выводит номер версии Python):

```
C:\code> py what.py                 # Командные строки запускающего модуля Windows
3.7.3

C:\code> py -2 what.py                       # Переключатель номера версии
2.7.3

C:\code> py -3.3 -i what.py -a -b -c         # Аргументы для py, python и сценария
3.3.0
>>> ^Z
```

Фактически, как показывает последний запуск в предыдущем примере, командные строки, использующие запускающий модуль, могут предоставлять аргументы для самого запускающего модуля (`-3.3`), интерпретатора Python (`-i`) и вашего сценария (`-a`, `-b` и `-c`). Запускающий модуль также способен разбирать номера версий из строк `#!` в стиле Unix, находящихся в начале файлов сценариев, что будет обсуждаться в следующем приложении.

Дополнительная помощь

В наши дни стандартный набор руководств включает ценные подсказки по применению интерпретатора Python на различных платформах. Стандартный набор руководств становится доступным после установки Python через меню Пуск (например, элемент Python 3.7 Manuals в группе Python 3.7), а также в онлайновом режиме на веб-сайте http://www.python.org. Поищите в наборе руководств раздел верхнего уровня, озаглавленный "Using the Python Interpreter" ("Использование интерпретатора Python"), в котором предлагаются дополнительные подсказки и советы, а также детали о межплатформенных переменных среды и командных строках.

Как обычно, веб-сеть тоже является вашим союзником, особенно в области, часто развивающейся быстрее, чем могут обновляться книги вроде этой. Учитывая широкое распространение Python, высока вероятность того, что вы сумеете найти в веб-сети ответы на все вопросы по высокоуровневому применению.

Приложение Б

Запускающий модуль Windows для Python

В этом приложении описан запускающий модуль Windows для Python, который автоматически устанавливается, начиная с версии Python 3.3, и доступен отдельно в веб-сети для использования с более старыми версиями. Он предоставляет дополнительный уровень кода, который выбирает и запускает установленный интерпретатор Python, обеспечивая согласованное выполнение программ, когда на одном компьютере сосуществует несколько версий Python.

Настоящее приложение рассчитано на программистов, работающих с Python в Windows. Несмотря на специфичную для платформы природу, его целевой аудиторией являются как новички в Python (большинство из них начинает именно в Windows), так и опытные разработчики на Python, которые пишут код, переносимый между Windows и Unix. Вы увидите, что данный запускающий модуль достаточно радикально изменяет правила в Windows, чтобы оказать влияние на *всех*, кто применяет Python в Windows или собирается заняться этим в будущем.

Наследие Unix

Чтобы полностью понять протоколы запускающего модуля, мы должны начать с краткого урока истории. Разработчики Unix давно продумали протокол для назначения программы, которая будет запускать код сценария. В системах Unix (включая Linux и Mac OS X) первая строка в текстовом файле сценария является особой, если она начинается с двухсимвольной последовательности: `#!`.

В главе 3 первого тома приводился краткий обзор данной темы, но здесь предлагается иной взгляд. В сценариях Unix такие строки задают программу, которая должна выполнить остальное содержимое сценария, указывая ее после `#!`.

Программа указывается с использованием либо полного пути к каталогу, где она находится, либо вызова Unix-утилиты `env`, которая ищет целевую программу в соответствии с `PATH` — настраиваемой переменной среды со списком каталогов для поиска в них исполняемых программ:

```
#!/usr/local/bin/python
...код сценария... # Запустить под управлением этой конкретной программы

#!/usr/bin/env python
...код сценария...  # Запустить под управлением программы python, найденной в PATH
```

Делая такой сценарий исполняемым (например, через chmod +x script.py), вы можете запускать его за счет ввода только имени файла сценария в командной строке; последовательность #! в начале файла затем направляет командную оболочку Unix на программу, которая выполнит остаток кода в файле. В зависимости от структуры установки платформы указанное после символов #! имя python может быть действительной исполняемой программой или символической ссылкой на исполняемую программу, расположенную где-то в другом месте. В строках подобного рода можно также явно указывать более конкретную исполняемую программу, такую как python3. В любом случае за счет изменения строк #!, символических ссылок либо настроек PATH у разработчиков в Unix есть возможность направлять сценарий подходящей установленной версии интерпретатора Python.

Конечно, ничего из описанного выше не относится к среде Windows, в которой строки #! не имеют какого-то особого смысла. Исторически сложилось так, что в Windows сам интерпретатор Python игнорировал такие строки, считая их комментариями (символ # начинает комментарий в языке). И все же идея выбора исполняемой программы интерпретатора Python для каждого файла представляет собой захватывающую возможность в мире, где версии Python 2.X и Python 3.X часто сосуществуют на одном компьютере. Учитывая тот факт, что в любом случае многие программисты предусматривают строки #! для переносимости в Unix, похоже, идея созрела для эмуляции.

Наследие Windows

В другом лагере модель установки очень сильно отличалась. В прошлом (точнее, до версии Python 3.3) установщик Windows обновлял глобальный реестр Windows так, что последняя версия интерпретатора Python, установленная на компьютере, и была той версией, которая открывала файлы Python, когда совершался двойной щелчок на их значках или запуск в командной строке по прямым именам.

Некоторые пользователи Windows могут опознать это как файловые *ассоциации*, конфигурируемые в окне Программы по умолчанию панели управления. В Windows нет необходимости делать файлы исполняемыми, как требуется для сценариев Unix. Фактически концепция подобного рода в Windows отсутствует — файловых ассоциаций и команд достаточно для того, чтобы запускать файлы в качестве программ.

В рамках такой модели установки, если вы хотите открыть файл с помощью версии, отличающейся от последней установленной, то должны предоставить в командной строке полный путь к нужному интерпретатору Python или вручную обновить файловые ассоциации для запуска желаемой версии. Можно было бы также направлять обобщенные команды python на конкретный интерпретатор Python, устанавливая либо изменяя настройку PATH, но это придется делать самостоятельно, вдобавок прием неприменим для запуска сценариев двойным щелчком на значках или в других контекстах.

Именно так выглядит естественный порядок в Windows (когда производится двойной щелчок на файле .doc, то обычно открывается последняя установленная версия редактора Word), и подобное положение вещей существовало с тех пор, как появился Python для Windows. Однако прием не настолько идеален при наличии сценариев Python, требующих разных версий на одном и том же компьютере — ситуация, которая становится все более распространенной и, пожалуй, даже обычной в эру двух линеек Python 2.X/Python 3.X. Запуск нескольких версий Python в среде Windows до выхода Python 3.3 был утомительным для разработчиков и обескураживающим для новоприбывших.

Введение в запускающий модуль Windows

Запускающий модуль Windows, который поставляется и автоматически устанавливается, начиная с версии Python 3.3, а также доступен в виде автономного пакета для использования с более ранними версиями, устраняет описанные выше недочеты предшествующей модели установки за счет предоставления двух новых исполняемых файлов:

- py.exe для консольных программ;
- pyw.exe для программ, отличающихся от консольных (обычно имеющих графический пользовательский интерфейс).

Указанные две исполняемые программы регистрируются для открытия соответственно файлов .py и .pyw через файловые ассоциации Windows. Подобно первоначальной основной программе python.exe (роль которой они не умаляют, но могут в значительной степени заместить ее) новые исполняемые программы также зарегистрированы для открытия файлов байт-кода, запускаемых напрямую. В рамках своих возможностей новые исполняемые программы:

- автоматически открывают файлы исходного кода и байт-кода Python, которые запускаются двойным щелчком на значках или вводом команд с именами файлов, через файловые ассоциации Windows;
- обычно устанавливаются в пути поиска системы и в случае применения в командной строке не требуют указания пути к каталогу или настройки переменной среды PATH;
- позволяют легко передавать номера версий Python как аргументы командной строки при запуске сценариев и интерактивных сеансов;
- разбирают строки комментариев #! в стиле Unix, находящиеся в начале сценариев, чтобы определить, какая версия интерпретатора Python должна использоваться для запуска кода внутри файла.

Совокупный эффект заключается в том, что при наличии нескольких версий Python в среде Windows благодаря запускающему модулю вы больше не ограничены версией, установленной последней, либо полными командными строками. Взамен вы можете выбирать версии явно на основе файлов и команд, а также указывать версии в частичной или полной форме в обоих контекстах. Ниже описано, как все работает.

1. Чтобы выбирать версии *для каждого файла*, помещайте в начало сценариев комментарии в стиле Unix следующего вида:

   ```
   #!python2
   #!/usr/bin/python2.7
   #!/usr/bin/env python3
   ```

2. Чтобы выбирать версии *для каждой команды*, применяйте командные строки в следующих формах:
   - `py -2 m.py`
   - `py -2.7 m.py`
   - `py -3 m.py`

Например, *первая* из указанных методик может служить своего рода директивой для объявления версии Python, на которую полагается сценарий, и которая будет применяться запускающим модулем всякий раз, когда сценарий запускается командной строкой или двойным щелчком на значке (ниже приведены вариации файла по имени script.py):

```
#!python3
...
...Сценарий Python 3.X...      # Выполняется под управлением последней
                               #   установленной версии Python 3.X
...
#!python2
...
...Сценарий Python 2.X...      # Выполняется под управлением последней
                               #   установленной версии Python 2.X
...
#!python2.6
...
...Сценарий Python 2.6...      # Выполняется под управлением
                               #   версии Python 2.6 (только)
...
```

В среде Windows команды вводятся в окне командной строки с подсказкой, обозначаемой здесь с помощью `C:\code>`. Первая команда ниже дает такой же результат, как вторая и двойной щелчок на значке, из-за файловых ассоциаций:

```
C:\code> script.py          # Выполняется в соответствии со строкой #! файла,
                            #   если она есть, иначе так, как принято по умолчанию
C:\code> py script.py       # То же самое, но py.exe запускается явно
```

В качестве альтернативы *вторая* методика из описанных выше позволяет выбирать версии с помощью аргументов командной строки:

```
C:\code> py -3 script.py  # Выполняется под управлением последней версии Python 3.X
C:\code> py -2 script.py  # Выполняется под управлением последней версии Python 2.X
C:\code> py -2.6 script.py # Выполняется под управлением версии Python 2.6 (только)
```

Прием работает при запуске как сценариев, так и интерактивного сеанса интерпретатора (когда сценарий не указан):

```
C:\code> py -3              # Запускает последнюю версию Python 3.X,
                            #   интерактивный сеанс
C:\code> py -2              # Запускает последнюю версию Python 2.X,
                            #   интерактивный сеанс
C:\code> py -3.1            # Запускает версию Python 3.1 (только),
                            #   интерактивный сеанс
C:\code> py                 # Запускает версию Python, принятую по умолчанию
```

При наличии строки `#!` в файле *и* аргумента с номером версии в командной строке версия, заданная в командной строке, аннулирует версию, указанную в директиве `#!` файла:

```
#! python3.2
...
...Сценарий Python 3.X...
...
C\code> py script.py        # Выполняется под управлением версии Python 3.2
                            #   в соответствии с директивой файла
C\code> py -3.1 script.py   # Выполняется под управлением версии Python 3.1,
                            #   даже если присутствует Python 3.2
```

Запускающий модуль также использует *эвристические правила* для выбора конкретной версии Python, когда она опущена или описана лишь частично. Скажем, в случае указания только 2 запускается последняя версия Python 2.X, а если версия опущена, то предпочтение отдается Python 3.X.

Явный выбор версии выглядит полезным дополнением для Windows, где многие (вероятно даже большинство) новоприбывших получают свое первое представление о языке, особенно в текущем двойственном мире Python 2.X/Python 3.X. Он делает возможным более элегантное сосуществование файлов Python 2.X и Python 3.X на одном компьютере и предлагает разумный подход к управлению версиями в командной строке.

Исчерпывающие сведения о запускающем модуле Windows, включая более продвинутые средства и сценарии использования, описание которых здесь либо сокращено, либо вообще опущено, ищите в пояснительной записке по выпуску Python и в документах PEP. Среди прочего запускающий модуль также позволяет выбирать между 32- и 64-разрядными установленными версиями, указывать принятые по умолчанию версии в конфигурационных файлах и определять специальное расширение командных строк `#!`.

Учебное руководство по запускающему модулю Windows

Читателям, знакомым с написанием сценариев в Unix, предыдущего раздела может оказаться достаточно, чтобы приступить к работе. Для других в настоящем разделе предоставляется дополнительный контекст в форме учебного руководства, демонстрирующего конкретные примеры запускающего модуля в действии, которые вы можете отследить. Тем не менее, попутно здесь также раскрываются дальнейшие детали о запускающем модуле, поэтому даже опытные пользователи Unix могут извлечь пользу от беглого просмотра раздела.

Для начала мы будем применять показанный далее простой сценарий `what.py`, который может запускаться под управлением Python 2.X и 3.X и выдает номер версии интерпретатора Python, выполняющего его код. В нем используется `sys.version` — строка, первый компонент которой после разделения по пробельным символам содержит номер версии Python:

```
#!python3
import sys
print(sys.version.split()[0])        # Первая часть строки
```

Наберите код данного сценария в предпочитаемом редакторе текстовых файлов, откройте окно командной строки и перейдите в каталог, где сохранили сценарий (вместо `C:\code` можно выбрать любой другой каталог по собственному усмотрению; подсказки по работе в Windows приводились в главе 3 первого тома).

Комментарий в первой строке сценария служит указанием требующейся версии Python; по соглашениям, принятым в Unix, он должен начинаться с `#!` и допускает наличие пробела перед `python3`. На компьютере установлены версии Python 2.7, 3,1, 3.2, 3.3 и 3.7; давайте посмотрим, какая версия запускается в случае модификации первой строки сценария в последующих разделах, и в ходе дела исследуем директивы файлов, командные строки и принятые стандарты.

Шаг 1: использование директив версий в файлах

В том виде, как сценарий реализован, при запуске двойным щелчком на значке или посредством командной строки первая строка предписывает зарегистрированному запускающему модулю `py.exe` выполнить его с применением последней установленной версии Python 3.X:

```
#! python3
import sys
print(sys.version.split()[0])

C:\code> what.py          # Выполняется в соответствии с директивой файла
3.7.3

C:\code> py what.py       # То же самое: последняя версия Python 3.X
3.7.3
```

Опять-таки пробел после #! необязателен; здесь он добавлен, чтобы удостовериться в этом. Обратите внимание, что первая команда запуска what.py эквивалентна двойному щелчку на значке и полному варианту py what.py, т.к. программа py.exe зарегистрирована для автоматического открытия файлов .py в файловых ассоциациях реестра Windows при установке запускающего модуля.

Также имейте в виду, что когда в документации по запускающему модулю (включая данное приложение) упоминается *последняя* версия, то под ней понимается версия с *наибольшим номером*. То есть речь идет о последней выпущенной версии, а не о версии, которая была установлена на компьютере позже всех (например, если вы установили Python 3.3 после Python 3.7, то #! python3 выберет Python 3.7). Запускающий модуль циклически проходит по имеющимся на компьютере версиям интерпретаторов Python, чтобы отыскать версию с наибольшим номером, которая соответствует вашей спецификации или принятым стандартам; подход отличается от предшествующей модели, где выигрывала версия, установленная позже всех.

Изменение имени в первой строке сценария на python2 приводит взамен к запуску установленной версии Python 2.X (в действительности с наибольшим номером):

```
#! python2
...остальной код сценария не изменился...

C:\code> what.py          # Выполняется с помощью последней версии Python 2.X
                          #   согласно #!
2.7.17
```

При необходимости можно запросить более конкретную версию интерпретатора – скажем, если вас не интересует последняя версия в линейке Python:

```
#! python3.1
...

C:\code> what.py          # Выполняется с помощью версии Python 3.1 согласно #!
3.1.4
```

Это верно даже когда запрошенная версия *не установлена*, что трактуется запускающим модулем как ошибочный случай:

```
#! python2.6
...

C:\code> what.py
Requested Python version (2.6) is not installed
Запрошенная версия Python (2.6) не установлена
```

Нераспознанные строки #! также рассматриваются как ошибки, если только вы не укажете номер версии в командной строке с целью корректировки, что более подробно описано в следующем разделе:

```
#!/bin/python
...

C:\code> what.py
```

```
Unable to create process using '/bin/python "C:\code\what.py" '
Не удалось создать процесс, используя /bin/python "C:\code\what.py"

C:\code> py what.py
Unable to create process using '/bin/python what.py'
Не удалось создать процесс, используя /bin/python what.py

C:\code> py -3 what.py
3.7.3
```

Формально запускающий модуль распознает строки `#!` в стиле Unix, находящиеся в начале файлов сценариев, которые соответствуют одной их следующих четырех схем:

```
#!/usr/bin/env python*
#!/usr/bin/python*
#!/usr/local/bin/python*
#!python*
```

Любая строка `#!`, не принимающая одну из таких распознаваемых и разбираемых форм, считается полностью заданной командной строкой для запуска процесса выполнения файла, которая передается Windows без изменений и приводит к выдаче показанного ранее сообщения об ошибке, если она не является допустимой командой Windows. (Запускающий модуль через свои конфигурационные файлы также поддерживает "настраиваемые" расширения команд, которые опробуются перед передачей нераспознанных команд среде Windows, но здесь мы их не затрагиваем.)

В распознаваемых строках `#!` пути к каталогам записываются в соответствии с соглашением Unix для обеспечения переносимости на эту платформу. Часть `*` в конце распознаваемых схем обозначает необязательный номер версии Python в одной из трех форм.

Неполный номер (например, `python3`***)***

Для запуска установленной версии, которая имеет наибольший младший номер выпуска среди версий с заданным старшим номером выпуска.

Полный номер (например, `python3.1`***)***

Для запуска только конкретной версии; можно указать необязательный суффикс `-32`, чтобы выбрать 32-разрядную версию (скажем, `python3.1-32`)

Номер опущен (например, `python`***)***

Для запуска версии, принятой запускающим модулем в качестве стандартной.

Файлы, вообще не содержащие строк `#!`, ведут себя точно так же, как файлы, в строках `#!` которых указано обобщенное имя `python` — вышеупомянутый вариант с отсутствием номера версии — и находятся под влиянием стандартных настроек `PY_PYTHON`. Первый случай, неполный номер, также может находиться под влиянием настроек среды, специфичных к версии (например, установите `PY_PYTHON3` в `3.1`, чтобы для `python3` выбиралась версия Python 3.1, и `PY_PYTHON2` в `2.6`, чтобы для `python2` выбиралась версия Python 2.6).

Однако важно отметить, что все, находящееся после части `*` в формате строки `#!`, трактуется как аргументы командной строки самого интерпретатора Python (т.е. программы `python.exe`), если только вы также не предоставите аргументы в командной строке `py`, которые запускающий модуль считает заменой аргументов в строке `#!`:

```
#!python3 [здесь находятся любые аргументы python.exe]
...
```

Сюда входят все аргументы командной строки Python, которые встречались в приложении А. Но это подводит нас к командным строкам запускающего модуля в целом и является достаточным основанием для плавного перехода к следующему разделу.

Шаг 2: использование переключателей версий командной строки

Как уже упоминалось, переключатель версии в командной строке можно применять для выбора версии интерпретатора Python, если она не указана в файле. В таком случае вы вводите командную строку py или pyw и передаете ей переключатель версии, а не полагаетесь на файловые ассоциации в реестре либо на предоставление версий в строке #! файла (или в дополнение к нему). Ниже мы удаляем директиву #! из нашего сценария:

```
# директива запускающего модуля отсутствует
...
C:\code> py -3 what.py      # Выполняется согласно переключателю командной строки
3.7.3

C:\code> py -2 what.py      # То же самое: последняя установленная версия Python 2.X
2.7.17

C:\code> py -3.2 what.py    # То же самое: конкретно (и только) версия Python 3.2
3.2.3

C:\code> py what.py         # Выполняется согласно принятой в запускающем
                            #  модуле стандартной версии
3.7.3
```

Но переключатели командной строки также имеют приоритет над указанием версии в директиве файла:

```
#! python3.1
...
C:\code> what.py            # Выполняется в соответствии с директивой файла
3.1.4

C:\code> py what.py         # То же самое
3.1.4

C:\code> py -3.2 what.py    # Переключатели аннулируют директивы
3.2.3

C:\code> py -2 what.py      # То же самое
2.7.17
```

Формально запускающий модуль принимает следующие виды аргументов командной строки (которые в точности отражают часть * в конце строки #! файла, описанную в предыдущем разделе):

```
-2          Запуск последней версии Python 2.X
-3          Запуск последней версии Python 3.X
-X.Y        Запуск указанной версии Python (X - это 2 или 3)
-X.Y-32     Запуск указанной 32-разрядной версии Python
```

И командные строки запускающего модуля принимают такую общую форму:

```
py [аргумент py.exe] [аргументы python.exe] script.py [аргументы script.py]
```

Все, что находится за собственным аргументом запускающего модуля (если он есть), трактуется как если бы оно передавалось программе python.exe — обычно

сюда входят любые аргументы для самого интерпретатора Python, за которыми следует имя файла сценария и любые аргументы, предназначенные для сценария.

Привычные формы спецификации программы -m *модуль*, -c *команда* и – тоже работают в командной строке py, как и все остальные аргументы командной строки Python, раскрытые в приложении А. Ранее упоминалось о том, что аргументы для python.exe также могут появляться в конце строки директивы #!, если они используются, хотя аргументы в командных строках py их аннулируют.

Чтобы увидеть, как все работает, давайте реализуем новый сценарий, который расширяет предыдущий отображением аргументов командной строки; sys.argv является собственными аргументами сценария, и вдобавок применяется переключатель -i интерпретатора Python (python.exe), обеспечивающий открытие интерактивной подсказки (>>>) после выполнения сценария:

```
# Файл args.py, отображает также собственные аргументы
import sys
print(sys.version.split()[0])
print(sys.argv)
C:\code> py -3 -i args.py -a 1 -b -c  # -3: py, -i: python, остальные: сценарий
3.7.3
['args.py', '-a', '1', '-b', '-c']
>>> ^Z
C:\code> py -i args.py -a 1 -b -c     # Аргументы для python и сценария
2.7.17
['args.py', '-a', '1', '-b', '-c']
>>> ^Z
C:\code> py -3 -c print(99)    # -3 для py, остальные для python: "-c команда"
99
C:\code> py -2 -c "print 99"
99
```

Обратите внимание, что первые две команды запускают стандартную версию Python, если только версия не задается в командной строке, поскольку в самом сценарии отсутствует строка #!.

Выводы: чистый выигрыш для Windows

Запускающий модуль, который поставляется, начиная с Python 3.3, предлагает согласованный способ управления сценариями и установленными интерпретаторами смешанных версий. Вы наверняка сочтете запускающий модуль ценным средством, как только приступите к его использованию.

В действительности у вас также может возникнуть желание начать вырабатывать привычку к написанию совместимых с Unix строк #! в своих сценариях для Windows, указывая явные номера версий (например, #!/usr/bin/python3). Это не только объявляет требования вашего кода и организует его надлежащее выполнение в Windows, но также обходит принятые по умолчанию версии в запускающем модуле и в будущем способно сделать сценарий пригодным в качестве исполняемой программы Unix.

Но вы должны отдавать себе отчет, что запускающий модуль может нарушить работу прежде допустимых сценариев, содержащих строки #!, требовать конфигурирования и внесения в код изменений того вида, на избегание которых он был нацелен. Новый шеф лучше старого, но похоже на то, что они ходили в одну и ту же школу.

За дополнительными сведениями о работе с интерпретатором в Windows обращайтесь в приложение А (установка и конфигурирование), главу 3 первого тома (общие концепции) и документацию, специфичную для платформы, внутри набора руководств по Python.

Приложение B

Изменения в Python и настоящая книга

В этом приложении кратко подытоживаются изменения, которые вносились в последние выпуски Python, организованные по изданиям книги, где они впервые затрагивались. Оно задумано как справочник для читателей предшествующих изданий и разработчиков, пришедших из предыдущих выпусков Python.

Вот как изменения в Python соотносятся с изданиями книги:

- в *пятом* издании (2013 год) раскрываются Python 3.3 и 2.7;
- в *четвертом* издании (2009 год) раскрываются Python 2.6 и 3.0 (с рядом возможностей Python 3.1);
- в *третьем* издании (2007 год) раскрывается Python 2.5;
- в *первом* и во *втором* изданиях (1999 и 2003 годы) раскрываются Python 2.0 и 2.2;
- в предшественнике настоящей книги, *Programming Python* (1996 год), раскрывался Python 1.3.

Следовательно, чтобы увидеть изменения, внесенные только в *пятое* издание, необходимо просмотреть перечисленные далее изменения в версиях Python 2.7, 3.2 и 3.3. Для выяснения, что изменилось в *четвертом и пятом* изданиях (т.е., начиная с *третьего*), также понадобится просмотреть изменения в версиях Python 2.6, 3.0 и 3.1. Изменения в третьем выпуске языка тоже описаны очень кратко, хотя теперь они, похоже, имеют лишь историческое значение.

Кроме того, имейте в виду, что приложение сосредоточено на крупных изменениях и их влиянии на книгу, но не предназначено служить исчерпывающим руководством по развитию Python. Более полные сведения об изменениях в каждом новом выпуске Python ищите в разделах "What's New" ("Что нового") стандартной документации, которая доступна по ссылке Documentation (Документация) на веб-сайте `python.org`. Документация и набор руководств обсуждались в главе 15 первого тома.

Главные отличия между Python 2.X и Python 3.X

Большая часть приложения связывает изменения в Python с освещением языка в книге. Если вас интересует краткий обзор самых заметных отличий между Python 2.X и Python 3.X, то приведенного далее материала может оказаться достаточно. Обратите внимание, что в этом разделе в основном сравниваются последние выпуски Python 3.X

и Python 2.X. Многие средства Python 3.X здесь не рассматриваются, потому что они либо были доставлены в Python 2.6 (например, оператор with и декораторы классов), либо позже перенесены в Python 2.7 (скажем, включения множеств и словарей), но не доступны в более ранних выпусках Python 2.X.

Отличия Python 3.X

Ниже приводится сводка по инструментам, которые отличаются между линейками Python.

- *Модель строк Unicode.* В Python 3.X обычные строки str поддерживают весь текст Unicode, включая ASCII, а отдельный тип bytes представляет низкоуровневые последовательности 8-битных байтов. В Python 2.X обычные строки str поддерживают 8-битный текст, включая ASCII, а отдельный тип unicode представляет обогащенный текст Unicode как вариант.
- *Модель файлов.* В Python 3.X файлы, создаваемые open, специализируются по содержимому — текстовые файлы задействуют кодировки Unicode и представляют содержимое в виде строк str, а двоичные файлы представляют содержимое как строки bytes. В Python 2.X файлы используют индивидуальные интерфейсы — файлы, создаваемые open, представляют содержимое в виде строк str для содержимого, которое является либо 8-битным текстом, либо байтовыми данными, и codecs.open задействует кодировки текста Unicode.
- *Модель классов.* В Python 3.X все классы автоматически становятся производными от object и обзаводятся многочисленными изменениями и расширениями классов *нового стиля*, включая отличающийся алгоритм наследования, координирование встроенных операций и порядок поиска MRO для деревьев с ромбовидными схемами. В Python 2.X нормальные классы следуют *классической* модели, а явное наследование от object или других встроенных типов запускает в работу модель нового стиля как вариант.
- *Встроенные итерируемые объекты.* В Python 3.X встроенные вызовы map, zip, range, filter, а также атрибуты keys, values и items словарей возвращают итерируемые объекты, которые генерируют значения по запросу. В Python 2.X все эти вызовы создают физические списки.
- *Вывод.* В Python 3.X предоставляется встроенная функция с ключевыми аргументами для конфигурирования, тогда как в Python 2.X предлагается оператор со специальным синтаксисом для конфигурирования.
- *Относительное импортирование.* В Python 2.X и Python 3.X поддерживаются операции относительного импортирования from ., но в Python 3.X изменилось правило поиска, чтобы пропускать собственный каталог пакета для нормальных операций импортирования.
- *Настоящее деление.* В Python 2.X и Python 3.X поддерживается операция деления с округлением в меньшую сторону //, но в Python 3.X операция / является настоящим делением и предохраняет дробные остатки, тогда как в Python 2.X операция / специфична к типу.
- *Целочисленные типы.* В Python 3.X есть единственный целочисленный тип, который поддерживает расширенную точность. В Python 2.X имеется тип int с нормальной и тип long с расширенной точностью, а также обеспечивается автоматическое преобразование в long.

- *Области видимости включений.* В Python 3.X все формы включений – списка, множества, словаря, генератор – локализуют переменные в выражении. В Python 2.X списковые включения этого не делают.
- *PyDoc.* Начиная с Python 3.2, поддерживается интерфейс с единым браузером `pydoc -b`, который стал обязательным в Python 3.3 и последующих версиях. В Python 2.X взамен может применяться первоначальный клиент с графическим пользовательским интерфейсом `pydoc -g`.
- *Хранение файлов с байт-кодом.* Начиная с версии Python 3.2, файлы с байт-кодом хранятся в подкаталоге `__pycache__` каталога с исходным кодом и получают имена, идентифицирующие версию. В Python 2.X файлы с байт-кодом хранятся в каталоге с исходным кодом и имеют обобщенные имена.
- *Встроенные системные исключения.* В версии Python 3.3 была переделана иерархия исключений для классов операционной системы и ввода-вывода, чтобы поддерживать дополнительные категории и детализацию. В Python 2.X иногда должны проверяться атрибуты исключений на предмет возникновения системных ошибок.
- *Сравнения и софтировки.* В Python 3.X сравнения по абсолютной величине для разнородных типов и словарей являются ошибками, а сортировки не поддерживают разнородные типы или универсальные функции сравнения. В Python 2.X все указанные формы работают.
- *Исключения на основе строк и функции модуля `string`.* Исключения на основе строк были удалены в Python 3.X, а также исчезли в Python 2.X, начиная с версии Python 2.6 (взамен должны использоваться классы). Функции модуля `string`, избыточные из-за наличия методов строковых объектов, также были удалены в Python 3.X.
- *Удаления в языке.* Как показано в табл. В.2, в Python 3.X удалены, переименованы или перемещены многих языковые элементы Python 2.X: `reload`, `apply`, `` `x` ``, `<>`, `0177`, `999L`, `dict.has_key`, `raw_input`, `xrange`, `file`, `reduce` и `file.xreadlines`.

Расширения, доступные только в Python 3.X

Ниже приведена сводка по инструментам, доступным только в Python 3.X.

- *Расширенные присваивания последовательностей.* В Python 3.X разрешено применять `*` в целях операций присваивания последовательностей для накопления в списке элементов итерируемых объектов, оставшихся несопоставленными. В Python 2.X похожих результатов можно добиться с помощью нарезания.
- *Оператор `nonlocal`.* В Python 3.X предлагается оператор `nonlocal`, который позволяет изменять имена из областей видимости объемлющих функций внутри вложенных функций. В Python 2.X похожих результатов можно добиться посредством атрибутов функций, изменяемых объектов и состояния классов.
- *Аннотации функций.* В Python 3.X предоставляется возможность аннотировать аргументы и возвращаемые типы функций с помощью объектов, которые хранятся в функции, но иным способом не используются. В Python 2.X похожих результатов часто можно достичь посредством дополнительных объектов или аргументов декораторов.

- *Аргументы с передачей только по ключевым словам.* В Python 3.X разрешено определение аргументов функций, которые обязаны передаваться как ключевые слова, обычно применяемые для добавочных конфигурационных параметров. В Python 2.X похожих результатов часто можно добиться с помощью анализа аргументов и извлечений из словаря.
- *Сцепленные исключения.* В Python 3.X посредством расширения `raise from` исключения можно объединять в цепочки, обеспечивая тем самым их присутствие в сообщениях об ошибках; начиная с версии Python 3.3, указание `None` позволяет аннулировать цепочку.
- *Оператор `yield from`.* Начиная с версии Python 3.3, оператор `yield` с помощью конструкции `from` может делегировать работу вложенному генератору. В Python 2.X посредством цикла `for` часто можно достичь похожих результатов для более простых сценариев использования.
- *Пакеты пространств имен.* Начиная с версии Python 3.3, модель пакетов была расширена, чтобы позволить пакетам охватывать множество каталогов без файла инициализации в качестве запасного варианта. В Python 2.X похожих результатов можно добиться с помощью расширений импортирования.
- *Запускающий модуль Windows.* Начиная с версии Python 3.3, вместе с интерпретатором поставляется запускающий модуль Windows, хотя он также доступен отдельно для применения с другими версиями Python, включая Python 2.X.
- *Внутреннее устройство.* Начиная с версии Python 3.2, многопоточная работа реализована с использованием временных интервалов вместо счетчиков команд виртуальной машины. Начиная с версии Python 3.3, текст Unicode сохраняется с применением схемы с переменной длиной, а не байтов фиксированного размера. Модель строк Python 2.X в целом сводит к минимуму использование Unicode.

Общие замечания: изменения в Python 3.X

Несмотря на то что линейка Python 3.X, раскрываемая в последних двух изданиях этой книги, представляет собой почти тот же самый язык, что и предшествующая ей линейка Python 2.X, в ряде ключевых областей существуют отличия. Как обсуждалось в предисловии к первому тому и подытоживалось в предыдущем разделе, не являющаяся факультативной модель Unicode, обязательные классы нового стиля и больший акцент на генераторах и других инструментах функционального программирования в линейке Python 3.X способны сделать работу с ней принципиально другой.

В общем и целом Python 3.X может считаться более *чистым* языком, но во многих отношениях он также и более *сложный* язык, опирающийся на значительно более развитые концепции. Фактически некоторые изменения предполагают, что для изучения Python вы уже должны знать Python. В предисловии к первому тому уминались самые заметные циклические зависимости внутри знаний в Python 3.X, которые влекут за собой прямые тематические зависимости.

Рассмотрим произвольный пример. Логическое обоснование для помещения словарных представлений внутрь вызова `list` в Python 3.X чрезвычайно тонкое и требует наличия солидных предварительных знаний — самое меньшее представлений, генераторов и протокола итерации. Ключевые аргументы аналогичным образом обязательны в простых инструментах (скажем, вывод, форматирование строк, создание словарей и сортировка), которые встречаются задолго до того, как новоприбывший получит достаточные знания о функциях, чтобы полностью понять их. Одной из це-

лей книги было оказание помощи в преодолении такой бреши в знаниях, присущей современному миру с двумя линейками Python 2.X и Python 3.X.

Изменения в библиотеках и инструментах

В Python 3.X были внесены дополнительные изменения, которые в настоящем приложении не рассматриваются просто потому, что они не оказывают влияния на книгу. Например, ряд стандартных библиотечных средств и инструментов разработки выходят за рамки тематики основного языка, обсуждаемой в книге, хотя некоторые из них упоминались попутно (скажем, `timeit`), а другие раскрывались здесь всегда (например, *PyDoc*).

Для полноты в последующих разделах отмечаются наработки Python 3.X в таких категориях. Позже в приложении будут описаны изменения, внесенные в данные категории, с указанием издания книги и версии Python, где они были представлены.

Изменения в стандартной библиотеке

Говоря формально, стандартная библиотека Python не является частью тематики основного языка, которой посвящена эта книга, хотя она всегда доступна с Python и буквально пронизывает реалистичные программы на Python. На самом деле библиотеки не подпадали под действие временного моратория на языковые изменения Python 3.X, установленного во время разработки версии Python 3.2.

Из-за этого изменения в стандартной библиотеке оказывают более сильное воздействие на книги, ориентированные на разработку прикладных приложений, вроде *Programming Python* (`http://shop.oreilly.com/product/9780596158118.do`). Хотя бóльшая часть функциональности стандартной библиотеки по-прежнему остается на месте, в Python 3.X продолжилось переименование модулей, группирование их в пакеты и изменение схем обращения к API-интерфейсам.

Однако некоторые изменения в библиотеке оказываются гораздо более широкими. Например, модель *Unicode* в Python 3.X становится причиной широко распространенных отличий стандартной библиотеки – она потенциально влияет на любую программу, обрабатывающую содержимое файлов, имена файлов, средства прохода по каталогам, каналы, дескрипторные файлы, сокеты, текст в графических пользовательских интерфейсах, протоколы Интернета, такие как FTP и электронная почта, сценарии CGI, веб-содержимое многих видов и даже ряд инструментов для постоянства, в том числе файлы DBM, `shelve` и `pickle`.

Более полный список изменений в стандартной библиотеке Python 3.X представлен в разделах "What's New" документации для выпусков Python 3.X (особенно Python 3.0). Из-за повсеместного применения Python 3.X книга *Programming Python* также может служить руководством по изменениям в стандартной библиотеке Python 3.X.

Изменения в инструментах

Хотя большинство инструментов разработки одинаковы в Python 2.X и Python 3.X (скажем, инструментов для отладки, профилирования, измерения времени и тестирования), в Python 3.X некоторые из них претерпели изменения вместе с языком и стандартной библиотекой. Например, система документирования модулей *PyDoc* отошла от своей прежней модели клиента с графическим пользовательским интерфейсом в Python 3.2 и предшествующих версиях, заменив ее интерфейсом с единым браузером.

В данной категории есть и другие изменения, заслуживающие внимания. Пакет `distutils`, используемый для распространения и установки стороннего программного обеспечения, в Python 3.X заменен новой системой *пакетирования*. Описанная

в книге новая схема хранения файлов с байт-кодом в __pycache__, хотя и является усовершенствованием, способна повлиять на многие инструменты Python и программы. Начиная с версии Python 3.2, внутренняя реализация *многопоточной обработки* изменилась для сокращения состязаний за счет модификации глобальной блокировки интерпретатора (GIL), чтобы применять временные интервалы вместо счетчика команд виртуальной машины.

Переход на Python 3.X

Если вы переходите из Python 2.X на Python 3.X, тогда обязательно ознакомьтесь со сценарием 2to3 для автоматического преобразования кода, который поставляется в составе Python 3.X. В текущий момент он доступен в подкаталоге Tools\Scripts каталога с установленной копией Python. Сценарий 2to3 не способен транслировать абсолютно все и пытается переводить главным образом код на основном языке – API-интерфейсы стандартной библиотеки Python 3.X могут дополнительно отличаться. Тем не менее, он делает приличную работу по преобразованию большей части кода на Python 2.X для выполнения под управлением Python 3.X.

Наоборот, программа обратного преобразования 3to2, в настоящее время доступная в области стороннего программного обеспечения, может транслировать большинство кода на Python 3.X для запуска в средах Python 2.X. В зависимости от имеющихся целей и ограничений программа 2to3 или 3to2 может оказаться полезной, если вы обязаны поддерживать код для обеих линеек Python; ищите в веб-сети детали, а также дополнительные инструменты и методики.

Можно также написать код, который выполняется *переносимым образом* в Python 2.X и Python 3.X, с использованием методик, представленных в книге – импортирование средств Python 3.X из модуля __future__, избегание специфичных для версии инструментов и т.д. Многие примеры, рассмотренные в книге, нейтральны к платформе. К ним относятся инструменты для оценки из главы 21 первого тома, инструменты для перезагрузки и форматирования из главы 25 первого тома, инструменты для вывода деревьев классов из главы 31, большинство крупных примеров декораторов из глав 38 и 39, шуточный сценарий в конце главы 41 и многое другое. До тех пор, пока вы понимаете основные языковые отличия Python 2.X/Python 3.X, реализация переносимого кода часто прямолинейна.

Если вас интересует написание кода для Python 2.X и Python 3.X, тогда взгляните также на six – библиотеку не зависящих от версии инструментов отображения и переименования, которая в текущее время расположена по ссылке https://pypi.org/project/six/.

Естественно, данный пакет не может скомпенсировать все отличия в семантике языка и API-интерфейсах библиотеки, и во многих случаях для получения преимуществ переносимости вам придется применять инструменты его библиотеки вместо чистого Python. Однако взамен ваши программы станут более нейтральными к версии.

Изменения в Python, относящиеся к пятому изданию: Python 2.7, 3.2, 3.3

Описанные ниже специфические изменения были внесены в линейки Python 2.X и Python 3.X после выхода в свет четвертого издания и отражены в пятом издании. В частности, здесь документируются касающиеся книги изменения в версиях Python 2.7, 3.2 и 3.3.

Изменения в Python 2.7

С технической точки зрения версия Python 2.7 по большей части заключает в себе обратный перенос нескольких средств Python 3.X, которые раскрывались в предыдущем издании книги, но в прошлом как средства, доступные только в Python 3.X. В пятом издании они представляются также как инструменты Python 2.7. В их числе:

- литералы множеств:
  `{1, 4, 2, 3, 4}`
- включения множеств и словарей:
  `{c * 4 for c in 'spam'}, {c: c * 4 for c in 'spam'}`
- словарные представления, добавленные в качестве необязательных методов:
  `dict.viewkeys(), dict.viewvalues(), dict.viewitems()`
- разделители в форме запятых и автоматическая нумерация в `str.format` (из Python 3.1):
  `'{:,.2f} {}'.format(1234567.891, 'spam')`
- вложенные операторы `with` диспетчеров контекста (из Python 3.1):
  `with X() as x, Y() as y: ...`
- улучшения в отображении `repr` объектов чисел с плавающей точкой (из Python 3.1: см. далее).

Чтобы выяснить, где эти темы раскрывались в книге, взгляните на их записи в списке изменений Python 3.0, представленном в табл. В.1, или в разделе, посвященном изменениям Python 3.1 далее в приложении. Они уже представлялись для Python 3.X, но были обновлены с целью отражения их доступности также в Python 2.7.

С точки зрения поддержки в соответствии с текущими планами Python 2.7 станет последним крупным выпуском линейки Python 2.X, но будет иметь длительный период сопровождения, в течение которого продолжит использоваться в производственной среде. После Python 2.7 все новые разработки должны делаться в линейке Python 3.X.

Тем не менее, невозможно предвидеть, выдержит ли такая официальная позиция проверку временем, принимая во внимание обширную пользовательскую базу Python 2.X. Дополнительные сведения ищите в начале первого тома; скажем, оптимизированная реализация PyPy по-прежнему доступна только для Python 2.X. Или, выражаясь в стиле группы "Монти Пайтон", "Я пока еще жив..." — следите за развитием линейки Python 2.X.

Изменения в Python 3.8

На момент выхода настоящей книги на русском языке стала доступной стабильная версия Python 3.8: `https://docs.python.org/3/whatsnew/3.8.html`.

Изменения в Python 3.7

Полные сведения об изменениях и нововведениях версии Python 3.7 доступны по ссылке `https://docs.python.org/3.7/whatsnew/3.7.html`.

Новые синтаксические средства:

- отложенная оценка аннотаций типов.

Обратно несовместимые изменения синтаксиса:

- `async` и `await` теперь являются зарезервированными ключевыми словами.

Новые библиотечные модули:

- `contextvars`, `dataclasses`, `importlib.resources`.

Новые встроенные средства:

- функция `breakpoint()`.

Усовершенствования модели данных Python:

- настройка доступа к атрибутам модулей.
- базовая поддержка модуля `typing` и обобщенных типов.
- свойство сохранения порядка вставки в объектах `dict` было объявлено официальной частью спецификации языка Python.

Значительные улучшения в стандартной библиотеке:

- модуль `asyncio` обзавелся новыми возможностями, став более удобным и производительным.
- модуль `time` получил поддержку для функций с наносекундной разрешающей способностью.

Кроме того, были внесены разнообразные усовершенствования в реализацию CPython, C API и документацию, а также заметно улучшена производительность во многих областях.

Ниже описаны дополнительные изменения в языке.

- Теперь функции можно передавать более 255 аргументов, а функция может иметь более 255 параметров.
- Методы `bytes.fromhex()` и `bytearray.fromhex()` теперь игнорируют все пробельные символы ASCII, а только пробелы.
- Типы `str`, `bytes` и `bytearray` получили поддержку нового метода `isascii()`, который можно применять для проверки, содержатся ли в строке или байтах только символы ASCII.
- `ImportError` теперь отображает имя модуля и путь `__file__` модуля, когда оператор `from ... import ...` терпит неудачу.
- Теперь поддерживается циклическое импортирование, влекущее за собой абсолютное импортирование с привязкой подмодуля к имени.
- `object.__format__(x, '')` теперь эквивалентно `str(x)`, а не `format(str(self), '')`.
- Для улучшения динамического создания трассировок стека `types.TracebackType` теперь создается в коде Python и атрибут `tb_next` объектов трассировки допускает запись.
- При использовании переключателя `-m` элемент `sys.path[0]` теперь расширяется до полного пути к начальному каталогу, а не остается пустым.
- Для отображения времени импортирования каждого модуля теперь можно применять новый параметр `-X importtime` или переменную среды `PYTHONPROFILEIMPORTTIME`.

В версии Python 3.7 также появились новые модули: `contextvars`, `dataclasses`, `importlib.resources`.

Изменения в Python 3.3

Версия Python 3.3 включает удивительно большое количество изменений. Некоторые из них совершенно не совместимы с кодом, написанным для предшествующих выпусков в линейке Python 3.X. Среди этих изменений запускающий модуль Windows, устанавливаемый как обязательная часть Python 3.3, обладает широким потенциалом нарушить работу существующих сценариев Python 3.X, выполняемых под управлением Windows.

Ниже приведен краткий перечень заслуживающих внимания изменений, внесенных в Python 3.3, вместе с местами, где они применялись в книге.

- Уменьшенный *объем занимаемой памяти*, который больше соответствует Python 2.X, главным образом благодаря схеме хранения строк переменной длины, а также системе словарей с разделяемыми ключами для атрибутов (см. главы 37 и 32).
- Новая *модель пакетов пространств имен*, где пакеты нового стиля могут охватывать множество каталогов и не требовать файлов `__init__.py` (см. главу 24 первого тома).
- Новый синтаксис для делегирования работы подгенераторам: `yield from...` (см. главу 20 первого тома).
- Новый синтаксис для подавления контекста исключения: `raise...from None` (см. главу 34).
- Новый синтаксис для принятия литеральной формы Unicode из Python 2.X, облегчающий миграцию: Python 3.3 теперь трактует литерал Unicode вида `u'xxxx'` из Python 2.X как нормальную строку `'xxxx'` по аналогии с тем, что Python 2.X трактует байтовый литерал `b'xxxx'` из Python 3.X как нормальную строку `'xxxx'` (см. главы 4 и 7 первого тома, а также главу 37).
- Модернизированные *иерархии исключений* операционной системы и ввода-вывода, которые предоставляют более инклюзивные общие суперклассы, а также новые подклассы для распространенных ошибок, способные избавить от необходимости проверять атрибуты объектов исключений (см. главу 35).
- Интерфейс с единым браузером для системы *документации PyDoc*, запускаемый через `pydoc -b`, заменяет предшествующий автономный клиент с графическим пользовательским интерфейсом, который был доступен через кнопку Пуск в Windows 7 и предыдущих версиях и вызывался посредством команды `pydoc -g` (см. главу 15 первого тома).
- Изменения, внесенные в давно существующие *стандартные библиотечные* модули, в числе которых `ftplib`, `time` и `email`, а также потенциально `distutils`; влияние на эту книгу: в Python 3.X модуль `time` имеет новые переносимые вызовы (см. главу 21 первого тома и главу 39).
- Реализация функции `__import__` в `importlib.__import__` отчасти для унификации и прояснения ее назначения (см. главы 22 и 25 первого тома).

- Новая возможность в установщике для Windows, которая обеспечивает расширение переменной среды `PATH` с целью включения каталога с копией Python 3.3 (см. приложения А и Б).
- Новый *запускающий модуль Windows*, который пытается интерпретировать строки `#!` в стиле Unix для организации запуска сценариев Python в Windows. Он позволяет делать явный выбор между версиями Python 2.X и Python 3.X как в `#!`, так и в новых командных строках `py` на основе файлов и команд (см. приложение Б).

Изменения в Python 3.2

Версия Python 3.2 продолжила развитие линейки Python 3.X. Она разрабатывалась во время действия моратория, запрещающего внесение изменений в основной язык Python 3.X, так что изменения были незначительными. Ниже представлен краткий обзор изменений в Python 3.2 вместе со ссылками, где они учитываются в пятом издании.

- Изменение модели хранения файлов с байт-кодом: `__pycache__` (см. главы 2 и 22 первого тома).
- Автоматическая кодировка модуля `struct` для строк была удалена (см. главу 9 первого тома и главу 37).
- Улучшенная поддержка со стороны самого языка разделения `str/bytes` в Python 3.X (не имеет отношения к книге).
- Вызов `cgi.escape` был перемещен в Python 3.2+ (не имеет отношения к книге).
- Изменение в многопоточной обработке: временн*ые* интервалы (не имеет отношения к книге).

Изменения в Python, относящиеся к четвертому изданию: Python 2.6, 3.0, 3.1

Четвертое издание книги было обновлено для охвата версий Python *3.0* и *2.6*, а также учета небольшого количества изменений, внесенных в Python *3.1*. Изменения в Python 3.0 и 3.1 применимы ко всем будущим выпускам в линейке Python 3.X, в том числе Python 3.3, а изменения в Python 2.6 также являются частью Python 2.7. Как упоминалось ранее, ряд изменений, описанных здесь как изменения в Python 3.X, позже перешли в Python 2.7 (например, литералы множеств, а также включения множеств и словарей).

Изменения в Python 3.1

Вдобавок к изменениям в Python 3.0 и 2.6, перечисленным в последующих разделах, незадолго до передачи в печать четвертое издание было также дополнено примечаниями о заметных расширениях в предстоящем выпуске Python 3.1, в числе которых:

- разделители в форме запятых и автоматическая нумерация в `str.format` (см. главу 7 первого тома);
- синтаксис с множеством диспетчеров контекста в операторах `with` (см. главу 34);
- новые методы для числовых объектов (см. главу 5 первого тома);
- (не раскрывалось вплоть до пятого издания) изменения в отображении объектов чисел с плавающей точкой (см. главы 4 и 5 первого тома).

Указанные выше темы раскрываются в пятом издании книги. Так как версия Python 3.1 ориентировалась в основном на оптимизацию и вышла сравнительно скоро после Python 3.0, четвертое издание также напрямую применимо к Python 3.1. Фактически из-за того, что Python 3.1 полностью замещает Python 3.0, и поскольку в любом случае лучше отдавать предпочтение последней версии Python, всякий раз, когда в четвертом издании упоминается Python 3.0, это относится к вариациям языка, которые были введены в Python 3.0, но присутствуют во всей линейке Python 3.X.

Есть одно заметное исключение: в четвертом издании *не* рассматривалась появившаяся в Python 3.1 схема отображения `repr` для чисел *с плавающей точкой*. Новый алгоритм отображения пытается выводить числа с плавающей точкой по возможности более интеллектуально, обычно с меньшим (но иногда бо́льшим) количеством десятичных цифр — изменение, которое отражено в пятом издании.

Изменения в Python 3.0 и 2.6

Изменения в языке, относящиеся к четвертому изданию, берут начало в версиях Python 3.0 и 2.6. Все изменения в Python 2.6 и многие изменения в Python 3.0 перешли в версии Python 2.7 и 3.3. Версия Python 2.7 была расширена некоторыми средствами Python 3.0, отсутствующими в Python 2.6, а версия Python 3.3 унаследовала все средства, введенные в Python 3.0.

Поскольку в первоначальном выпуске Python 3.X было слишком много изменений, они лишь кратко отмечаются в таблицах вместе со ссылками на дополнительные детали в книге. В табл. В.1 описан первый набор изменений в Python 3.X, раскрытых в четвертом издании, а также указаны главы в пятом издании, где они упоминаются.

Таблица В.1. Расширения в Python 2.6 и 3.0

Расширение	Глава (главы), где рассматривается
Функция `print` в Python 3.0	11 (первый том)
Оператор `nonlocal x,y` в Python 3.0	17 (первый том)
Метод `str.format` в Python 2.6 и 3.0	7 (первый том)
Строковые типы в Python 3.0: `str` для текста Unicode, `bytes` для двоичных данных	7 (первый том), 37
Разграничение между текстовыми и двоичными файлами в Python 3.0	9 (первый том), 37
Декораторы классов в Python 2.6 и 3.0: `@private('age')`	32, 39
Новые итераторы в Python 3.0: `range`, `map`, `zip`	14, 20 (первый том)
Словарные представления в Python 3.0: `D.keys`, `D.values`, `D.items`	8, 14 (первый том)
Операции деления в Python 3.0: с остатком, / и //	5 (первый том)
Литералы множеств в Python 3.0: `{a, b, c}`	5 (первый том)
Включения множеств в Python 3.0: `{x**2 for x in seq}`	4, 5, 14, 20 (первый том)
Включения словарей в Python 3.0: `{x: x**2 for x in seq}`	4, 8, 14, 20 (первый том)
Поддержка строк с двоичными цифрами в Python 2.6 и 3.0: `0b0101`, `bin(I)`	5 (первый том)

Окончание табл. В.1

Расширение	Глава (главы), где рассматривается
Тип для дробных чисел в Python 2.6 и 3.0: `Fraction(1, 3)`	5 (первый том)
Аннотации функций в Python 3.0: `def f(a:99, b:str)->int`	19 (первый том)
Аргументы с передачей только по ключевым словам в Python 3.0: `def f(a, *b, c, **d)`	18, 20 (первый том)
Расширенная распаковка последовательностей в Python 3.0: `a, *b = seq`	11, 13 (первый том)
Синтаксис относительного импортирования для пакетов, доступный в Python 3.0: `from .`	24 (первый том)
Диспетчеры контекста, доступные в Python 2.6 и 3.0: `with/as`	34, 36
Изменения синтаксиса исключений в Python 3.0: `raise`, `except/as`, суперкласс	34, 35
Сцепление исключений в Python 3.0: `raise e2 from e1`	34
Изменения зарезервированных слов в Python 2.6 и 3.0	11 (первый том)
Переход на классы нового стиля в Python 3.0	32
Декораторы свойств в Python 2.6 и 3.0: `@property`	38
Использование дескрипторов в Python 2.6 и 3.0	32, 38
Использование метаклассов в Python 2.6 и 3.0	32, 40
Поддержка абстрактных базовых классов в Python 2.6 и 3.0	29

Удаления в языке Python 3.0

В дополнение к расширениям некоторые инструменты Python 2.X были удалены из Python 3.X в попытке привести в порядок лежащее в его основе проектное решение. В табл. В.2 приведена сводка по удалениям в Python 3.X, которые оказали влияние на книгу. Как видно в табл. В.2, многие удаления в Python 3.X имеют прямые замены, часть которых также доступна в Python 2.6 и 2.7 для поддержки будущего перехода на Python 3.X.

Таблица В.2. Удаления в Python 3.0, влияющие на эту книгу

Удаление	Замена	Глава (главы), где рассматривается
`reload(M)`	`imp.reload(M)` (или `exec`)	3, 23 (первый том)
`apply(f, ps, ks)`	`f(*ps, **ks)`	18 (первый том)
`` `X` ``	`repr(X)`	5 (первый том)
`X <> Y`	`X != Y`	5 (первый том)
`long`	`int`	5 (первый том)
`9999L`	`9999`	5 (первый том)
`D.has_key(K)`	`K in D (or D.get(key) != None)`	8 (первый том)

Продолжение табл. В.2

Удаление	Замена	Глава (главы), где рассматривается
raw_input	input	3, 10 (первый том)
Старая функция input	eval(input())	3 (первый том)
xrange	range	13, 14 (первый том)
file	open (и классы модуля io)	9 (первый том)
X.next	X.__next__, вызываемый next(X)	14, 20 (первый том), 30
X.__getslice__	X.__getitem__ с передачей объекта среза	7 (первый том), 30
X.__setslice__	X.__setitem__ с передачей объекта среза	7 (первый том), 30
reduce	functools.reduce (или код цикла)	14, 19 (первый том)
execfile(filename)	exec(open(filename).read())	3 (первый том)
exec open(filename)	exec(open(filename).read())	3 (первый том)
0777	0o777	5 (первый том)
print x, y	print(x, y)	11 (первый том)
print >> F, x, y	print(x, y, file=F)	11 (первый том)
print x, y,	print(x, y, end=' ')	11 (первый том)
u'ccc' (вернулись в Python 3.3)	'ccc'	4, 7 (первый том), 37
'bbb' для байтовых строк	b'bbb'	4, 7, 9 (первый том), 37
raise E, V	raise E(V)	33, 34, 35
except E, X:	except E as X:	33, 34, 35
def f((a, b)):	def f(x): (a, b) = x	11, 18, 20 (первый том)
file.xreadlines	for line in file: (or X=iter(file))	13, 14 (первый том)
D.keys() и т.д. как списки	list(D.keys()) (словарные представления)	8, 14 (первый том)
map(), range() и т.д. как списки	list(map()), list(range()) (встроенные функции)	14 (первый том)
map(None, ...)	zip (или написанный вручную код для дополнения результатов)	13, 20 (первый том)
X=D.keys();X.sort()	sorted(D) (или list(D.keys()))	4, 8, 14 (первый том)
cmp(x, y)	(x > y) - (x < y)	30
X.__cmp__(y)	__lt__, __gt__, __eq__ и т.д.	30
X.__nonzero__	X.__bool__	30
X.__hex__, X.__oct__	X.__index__	30

Удаление	**Замена**	**Глава (главы), где рассматривается**
Функции сравнения при сортировке	Использование `key=transform` или `reverse=True`	8 (первый том)
<, >, <=, >= для словарей	Сравнение `sorted(D.items())` (или код цикла)	8, 9 (первый том)
`types.ListType`	`list` (`types` только для не встроенных имен)	9 (первый том)
`__metaclass__ = M`	`class C(metaclass=M):`	29, 32, 40
`__builtin__`	`builtins` (переименован)	17 (первый том)
`Tkinter`	`tkinter` (переименован)	18, 19, 25 (первый том), 30, 31
`sys.exc_type`, `exc_value`	`sys.exc_info()[0], [1]`	35, 36
`function.func_code`	`function.__code__`	19 (первый том), 39
`__getattr__`, запускаемый встроенными операциями	Методы `__X__` в классах оболочек	31, 38, 39
Переключатели командной строки `-t`, `-tt`	Несогласованное использование табуляций/пробелов всегда является ошибкой	10, 12 (первый том)
`from ... *`, внутри функции	Может находиться только на верхнем уровне файла	23 (первый том)
`import mod`, в том же самом пакете	`from . import mod`, форма относительно пакета	24 (первый том)
`class MyException:`	`class MyException(Exception):`	35
Модуль `exceptions`	Встроенная область видимости, руководство по библиотеке	35
Модули `thread`, `Queue`	`_thread`, `queue` (оба переименованы)	17 (первый том)
Модуль `anydbm`	`dbm` (переименован)	28
Модуль `cPickle`	`_pickle` (переименован, используется автоматически)	9 (первый том)
`os.popen2/3/4`	`subprocess.Popen` (`os.popen` сохранен)	14 (первый том)
Исключения на основе строк	Исключения на основе классов (также обязательны в Python 2.6)	33, 34, 35
Функции модуля `string`	Методы строкового объекта	7 (первый том)
Несвязанные методы	Функции (`staticmethod` для вызова через экземпляр)	31, 32
Сравнения и сортировки для смешанных типов	Сравнения по абсолютной величине (и сортировки) для нечисловых смешанных типов являются ошибками	5, 9 (первый том)

Изменения в Python, относящиеся к третьему изданию: Python 2.3, 2.4, 2.5

Третье издание книги было основательно обновлено, чтобы учесть версию *Python 2.5* и все изменения, внесенные в язык с момента публикации второго издания в конце 2003 года. (Второе издание было основано главным образом на версии *Python 2.2* с рядом средств Python 2.3, реализованных в конце проекта.) Вдобавок там, где было уместно, обсуждались ожидаемые изменения в приближающемся тогда выпуске Python 3.0. Ниже перечислены основные темы, связанные с языком, которые были раскрыты или расширены (номера глав соответствуют пятому изданию):

- новое условное выражение `B if A else C` (см. главы 12 и 19 первого тома);
- диспетчеры контекста `with/as` (см. главу 34);
- объединение `try/except/finally` (см. главу 34);
- синтаксис относительного импортирования (см. главу 24 первого тома);
- генераторные выражения (см. главу 20 первого тома);
- новые возможности генераторных функций (см. главу 20 первого тома);
- декораторы функций (см. главы 32 и 39);
- тип множества (см. главу 5 первого тома);
- новые встроенные функции: `sorted`, `sum`, `any`, `all`, `enumerate` (см. главы 13 и 14 первого тома);
- объекты десятичных чисел с фиксированной точностью (см. главу 5 первого тома);
- файлы, списковые включения и итераторы (см. главы 14 и 20 первого тома);
- новые инструменты для разработки: Eclipse, `distutils`, `unittest` и `doctest`, расширения IDLE, Shed Skin и т.д. (см. главу 2 первого тома и главу 36).

Повсюду в книге упоминаются менее значительные изменения в языке (скажем, широко распространенное применение `True` и `False`, новая функция `sys.exc_info` для извлечения деталей об исключении, а также избавление от исключений на основе строк, строковых методов и встроенных функций `apply` и `reduce`). В третьем издании также приводился расширенный обзор средств, которые были новыми во втором издании, включая срезы с тремя пределами и заменивший `apply` синтаксис вызова с произвольными аргументами.

Более ранние и более поздние изменения в Python

В каждом издании, предшествующем третьему, также учитывались изменения в Python. В первых двух изданиях (1999 и 2003 годов) раскрывались версии Python 2.0 и 2.2. В вышедшей в 1996 году книге *Programming Python, 1st Edition* все начиналось с версии Python 1.3, но детали здесь не приводятся из-за того, что теперь это старая история (во всяком случае, в переводе на сферу компьютеров).

Хотя предсказать будущее невозможно, с учетом проверки временем вполне вероятно, что основные идеи, подчеркнутые в настоящей книге, будут применимы также к будущим выпускам Python.

Приложение Г

Решения упражнений, приводимых в конце частей

Часть VI, "Классы и объектно-ориентированное программирование"

Упражнения приведены в главе 32.

1. **Наследование.** Ниже представлен код решения этого упражнения (файл adder.py) вместе с несколькими интерактивными тестами. Перегруженная операция `__add__` должна появляться только один раз в суперклассе, т.к. она вызывается специфичными к типам методами add в подклассах:

```
class Adder:
    def add(self, x, y):
        print('not implemented!')            # не реализован
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):                # Или в подклассах?
        return self.add(self.data, other)    # Или возвращаемый тип?

class ListAdder(Adder):
    def add(self, x, y):
        return x + y

class DictAdder(Adder):
    def add(self, x, y):
        new = {}
        for k in x.keys(): new[k] = x[k]
        for k in y.keys(): new[k] = y[k]
        return new

% python
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
>>> x = ListAdder()
```

```
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1:1}, {2:2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
В Python 3.3 и последующих версиях:
TypeError: can only concatenate list (not "ListAdder") to list
Ошибка типа: конкатенация возможна только списка (не ListAdder) со списком
В более ранних версиях Python:
TypeError: __add__ nor __radd__ defined for these operands
Ошибка типа: для этих операндов не определены ни __add__, ни __radd__
```

В последнем тесте обратите внимание на то, что выражения, где экземпляр класса появляется справа от операции +, вызывают ошибку; если вы хотите исправить это, используйте методы `__radd__`, как было описано в главе 30.

Если вы каким-либо образом сохраняете значение в экземпляре, тогда можете также переопределить метод `add` для приема только одного аргумента в духе других примеров в данной части книги (файл `adder2.py`):

```
class Adder:
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):      # Передавать одиночный аргумент
        return self.add(other)     # То, что слева от операции - это self
    def add(self, y):
        print('not implemented!')

class ListAdder(Adder):
    def add(self, y):
        return self.data + y

class DictAdder(Adder):
    def add(self, y):
        d = self.data.copy()  # Изменить для использования self.data вместо x
        d.update(y)           # Или "смошенничать" путем использования
        return d              # более быстрой встроенной функции

x = ListAdder([1, 2, 3])
y = x + [4, 5, 6]
print(y)                      # Выводится [1, 2, 3, 4, 5, 6]

z = DictAdder(dict(name='Bob')) + {'a':1}
print(z)                      # Выводится {'name': 'Bob', 'a': 1}
```

Поскольку значения присоединяются к объектам, а не передаются, данная версия, вероятно, является более объектно-ориентированной. И как только вы осознаете это, то возможно обнаружите, что можете вообще избавиться от метода `add` и просто определить специфичные для типов методы `__add__` в двух подклассах.

2. **Перегрузка операций.** В коде решения (файл `mylist.py`) применяется несколько методов перегрузки операций, исследованных в главе 30. Важно сделать копию начального значения в конструкторе, потому что оно может быть изменяемым объектом, ведь вы не хотите изменять либо иметь ссылку на объект, который возможно совместно используется где-то за пределами класса. Метод `__getattr__` направляет вызовы внутреннему списку. Подсказки о более простом способе реализации этого в Python 2.2 и последующих версиях ищите в разделе "Расширение типов путем создания подклассов" главы 32:

```
class MyList:
   def __init__(self, start):
      # self.wrapped = start[:]      # Копировать start, чтобы не было
                                     #   побочных эффектов
      self.wrapped = list(start)     # Обеспечить здесь наличие списка
   def __add__(self, other):
      return MyList(self.wrapped + other)
   def __mul__(self, time):
      return MyList(self.wrapped * time)
   def __getitem__(self, offset):   # В Python 3.X также передается срез
      return self.wrapped[offset]    # Для итерирования, если нет __iter__
   def __len__(self):
      return len(self.wrapped)
   def __getslice__(self, low, high): # В Python 3.X игнорируется:
                                     #   используется __getitem__
      return MyList(self.wrapped[low:high])
   def append(self, node):
      self.wrapped.append(node)
   def __getattr__(self, name):      # Остальные методы: сортировка/
                                     #   обращение порядка/и т.д.
      return getattr(self.wrapped, name)
   def __repr__(self):               # Обобщенный метод отображения
      return repr(self.wrapped)

if __name__ == '__main__':
   x = MyList('spam')
   print(x)
   print(x[2])
   print(x[1:])
   print(x + ['eggs'])
   print(x * 3)
   x.append('a')
   x.sort()
   print(' '.join(c for c in x))
```

```
c:\code> python mylist.py
['s', 'p', 'a', 'm']
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 's', 'p', 'a', 'm', 's', 'p', 'a', 'm']
a a m p s
```

Обратите внимание, что здесь важно копировать начальное значение (`start`) путем вызова `list`, а не нарезания, поскольку иначе результат может оказаться не настоящим списком и потому не реагировать на ожидаемые списковые мето-

ды, такие как append (например, нарезание строки возвращает другую строку, а не список). Вы могли бы копировать начальное значение MyList за счет нарезания, потому что его класс перегружает операцию нарезания и предоставляет ожидаемый списковый интерфейс; однако, вы должны избегать копирования, основанного на нарезании, для таких объектов, как строки.

3. **Создание подклассов.** Мое решение показано ниже (mysub.py). Ваше решение должно быть похожим:

```
from mylist import MyList

class MyListSub(MyList):
    calls = 0                                # Разделяется экземплярами
    def __init__(self, start):
        self.adds = 0                        # Отличается в каждом экземпляре
        MyList.__init__(self, start)

    def __add__(self, other):
        print('add: ' + str(other))
        MyListSub.calls += 1                 # Счетчик на уровне класса
        self.adds += 1                       # Счетчик для каждого экземпляра
        return MyList.__add__(self, other)

    def stats(self):
        return self.calls, self.adds         # Счетчик всех обращений к
                                             #  операции + и счетчик экземпляра

if __name__ == '__main__':
    x = MyListSub('spam')
    y = MyListSub('foo')
    print(x[2])
    print(x[1:])
    print(x + ['eggs'])
    print(x + ['toast'])
    print(y + ['bar'])
    print(x.stats())

c:\code> python mysub.py
a
['p', 'a', 'm']
add: ['eggs']
['s', 'p', 'a', 'm', 'eggs']
add: ['toast']
['s', 'p', 'a', 'm', 'toast']
add: ['bar']
['f', 'o', 'o', 'bar']
(3, 2)
```

4. **Методы атрибутов.** Ниже представлено мое решение. Обратите внимание, что в классических классах Python 2.X операции пытаются извлекать атрибуты также через __getattr__; вам необходимо возвращать значение, чтобы заставить это работать. Как отмечалось в главе 32 и в других местах, метод __getattr__ не вызывается для встроенных операций в Python 3.X (и в случае применения классов нового стиля в Python 2.X), так что выражения здесь вообще не перехватываются; в классах нового стиля класс подобного рода обязан явно переопределять методы перегрузки операций __X__. Ситуация более подробно обсуждалась в главах 28, 31, 32, 38 и 39: это может повлиять на большинство кода!

```
c:\code> py -2
>>> class Attrs:
        def __getattr__(self, name):
           print('get %s' % name)
        def __setattr__(self, name, value):
           print('set %s %s' % (name, value))

>>> x = Attrs()
>>> x.append
get append
>>> x.spam = 'pork'
set spam pork
>>> x + 2
get __coerce__
TypeError: 'NoneType' object is not callable
Ошибка типа: объект NoneType не является вызываемым
>>> x[1]
get __getitem__
TypeError: 'NoneType' object is not callable
Ошибка типа: объект NoneType не является вызываемым
>>> x[1:5]
get __getslice__
TypeError: 'NoneType' object is not callable
Ошибка типа: объект NoneType не является вызываемым

c:\code> py -3
>>> ...тот же самый начальный код...
>>> x + 2
TypeError: unsupported operand type(s) for +: 'Attrs' and 'int'
Ошибка типа: неподдерживаемые типы операндов для +: Attrs и int
>>> x[1]
TypeError: 'Attrs' object does not support indexing
Ошибка типа: объект Attrs не поддерживает индексирование
>>> x[1:5]
TypeError: 'Attrs' object is not subscriptable
Ошибка типа: объект Attrs не допускает индексацию
```

5. **Объекты множества.** Ниже приведен пример взаимодействия, которое вы должны получить. В комментариях указано, какие методы вызываются. Кроме того, обратите внимание, что в наши дни множества являются встроенным типом Python, поэтому реализация по большому счету может считаться лишь упражнением (дополнительные сведения о множествах ищите в главе 5 первого тома).

```
% python
>>> from setwrapper import Set
>>> x = Set([1, 2, 3, 4])          # Запускается __init__
>>> y = Set([3, 4, 5])

>>> x & y                          # __and__, пересечение, __repr__
Set:[3, 4]
>>> x | y                          # __or__, объединение, __repr__
Set:[1, 2, 3, 4, 5]

>>> z = Set("hello")               # __init__ удаляет дубликаты
>>> z[0], z[-1], z[2:]             # __getitem__
('h', 'o', ['l', 'o'])
```

```
>>> for c in z: print(c, end=' ')      # __iter__ (иначе __getitem__)
                                       #  [print из Python 3.X]
...
h e l o
>>> ''.join(c.upper() for c in z)      # __iter__ (иначе __getitem__)
'HELO'
>>> len(z), z                          # __len__, __repr__
(4, Set:['h', 'e', 'l', 'o'])
>>> z & "mello", z | "mello"
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm'])
```

Мое решение для расширения с множеством операндов выглядит следующим образом (файл `multiset.py`). В первоначальном множестве потребовалось заменить только два метода. Строка документации класса объясняет его работу.

```
from setwrapper import Set

class MultiSet(Set):
"""
Наследует все имена Set, но расширяет intersect и union для поддержки
множества операндов;
обратите внимание, что self - по-прежнему первый аргумент (теперь
хранящийся в *args);
кроме того, унаследованные операции & и | здесь вызывают новые методы с
2 аргументами, но обработка более 2 аргументов требует вызова метода, не
выражения; intersect не удаляет дубликаты: это делает конструктор Set.
"""
def intersect(self, *others):
    res = []
    for x in self:                     # Просмотр первой последовательности
        for other in others:           # Для всех оставшихся аргументов
            if x not in other: break   # Элемент в каждой последовательности?
        else:                          # Нет: выйти из цикла
            res.append(x)              # Да: добавить элемент в конец
    return Set(res)

def union(*args):                      # self - это args[0]
    res = []
    for seq in args:                   # Для всех аргументов
        for x in seq:                  # Для всех узлов
            if not x in res:
                res.append(x)          # Добавить новые элементы к результату
    return Set(res)
```

Взаимодействие с расширением будет выглядеть примерно так, как показано ниже. Обратите внимание, что пересечение можно выполнять с использованием `&` или вызова `intersect`, но для трех и более операндов потребуется вызывать `intersect`; `&` является бинарной операцией. Кроме того, имейте в виду, что мы могли бы назвать `MultiSet` просто `Set`, чтобы сделать изменение более прозрачным, если для ссылки на оригинал внутри `multiset` применяется `setwrapper.Set` (при желании класс можно было бы переименовать в конструкции `as` оператора `import`):

```
>>> from multiset import *
>>> x = MultiSet([1, 2, 3, 4])
>>> y = MultiSet([3, 4, 5])
>>> z = MultiSet([0, 1, 2])
```

```
>>> x & y, x | y                              # Два операнда
(Set:[3, 4], Set:[1, 2, 3, 4, 5])
>>> x.intersect(y, z)                         # Три операнда
Set:[]
>>> x.union(y, z)
Set:[1, 2, 3, 4, 5, 0]
>>> x.intersect([1,2,3], [2,3,4], [1,2,3])    # Четыре операнда
Set:[2, 3]
>>> x.union(range(10))          # Экземпляры не MultiSets тоже работают
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]
>>> w = MultiSet('spam')                      # Строковые множества
>>> w
Set:['s', 'p', 'a', 'm']
>>> ''.join(w | 'super')
'spamuer'
>>> (w | 'super') & MultiSet('slots')
Set:['s']
```

6. **Связи деревьев классов.** Вот как я изменил подмешиваемые классы `lister.py` и повторно прогнал тест, чтобы показать их формат. Сделайте то же самое для версии на основе `dir` и также при форматировании объектов класса в варианте подъема по дереву:

```
class ListInstance:
    def __attrnames(self):
        ...без изменений...

    def __str__(self):
        return '<Instance of %s(%s), address %s:\n%s>' % (
                        self.__class__.__name__,     # Имя класса
                        self.__supers(),   # Собственные суперклассы класса
                        id(self),                    # Адрес
                        self.__attrnames())          # Список имя=значение

    def __supers(self):
        names = []
        for super in self.__class__.__bases__:  # На один уровень выше класса
            names.append(super.__name__)        # name, не str(super)
        return ', '.join(names)

    # Или ', '.join(super.__name__ for super in self.__class__.__bases__)

c:\code> py listinstance-exercise.py
<Instance of Sub(Super, ListInstance), address 43671000:
        data1=spam
        data2=eggs
        data3=42
>
```

7. Композиция. Мое решение представлено далее (файл `lunch.py`) с описанием в комментариях к коду. Это один из случаев, когда проще выразить проблему в коде на Python, чем с помощью естественного языка:

```
class Lunch:
    def __init__(self):              # Создать/внедрить Customer, Employee
        self.cust = Customer()
        self.empl = Employee()
```

```
    def order(self, foodName):          # Начать эмуляцию заказа
                                        #   экземпляром Customer
        self.cust.placeOrder(foodName, self.empl)
    def result(self):                   # Запросить у экземпляра Customer
                                        #   название его блюда
        self.cust.printFood()

class Customer:
    def __init__(self):                 # Инициализировать блюдо с помощью None
        self.food = None
    def placeOrder(self, foodName, employee):  # Разместить заказ для
                                               #   экземпляра Employee
        self.food = employee.takeOrder(foodName)
    def printFood(self):                # Вывести название блюда
        print(self.food.name)

class Employee:
    def takeOrder(self, foodName):      # Возвратить блюдо с желаемым названием
        return Food(foodName)

class Food:
    def __init__(self, name):           # Сохранить название блюда
        self.name = name

if __name__ == '__main__':
    x = Lunch()                         # Код самотестирования
    x.order('burritos')                 # Если запускается, не импортируется
    x.result()
    x.order('pizza')
    x.result()
```

```
% python lunch.py
burritos
pizza
```

8. **Иерархия для представления животных в зоопарке.** Вот как я реализовал такую иерархическую классификацию на Python (файл `zoo.py`); это искусственный, но общепринятый кодовый шаблон, применимый ко многим реальным структурам, от графических пользовательских интерфейсов и баз данных сотрудников до космических аппаратов. Обратите внимание, что ссылка `self.speak` в `Animal` инициирует независимый поиск при наследовании, который находит `speak` в подклассе. Протестируйте реализацию интерактивно в соответствии с описанием упражнения. Попробуйте расширить иерархию новыми классами и создавать разнообразные классы в дереве.

```
class Animal:
    def reply(self): self.speak()       # Обратно в подкласс
    def speak(self): print('spam')      # Специальное сообщение

class Mammal(Animal):
    def speak(self): print('huh?')

class Cat(Mammal):
    def speak(self): print('meow')

class Dog(Mammal):
    def speak(self): print('bark')

class Primate(Mammal):
    def speak(self): print('Hello world!')

class Hacker(Primate): pass             # Унаследован от Primate
```

9. **Скетч "Мертвый попугай".** Ниже приведена моя реализация (файл parrot.py). Обратите внимание на то, как работает метод line в суперклассе Actor: за счет доступа к атрибутам self два раза он дважды направляет интерпретатор Python обратно на экземпляр и потому инициирует два поиска при наследовании – self.name и self.says() находят информацию в специфических подклассах:

```
class Actor:
   def line(self): print(self.name + ':', repr(self.says()))
class Customer(Actor):
   name = 'customer'
   def says(self): return "that's one ex-bird!"
class Clerk(Actor):
   name = 'clerk'
   def says(self): return "no it isn't..."
class Parrot(Actor):
   name = 'parrot'
   def says(self): return None
class Scene:
   def __init__(self):
      self.clerk = Clerk()          # Внедрить несколько экземпляров
      self.customer = Customer()    # Scene является смесью
      self.subject = Parrot()
   def action(self):
      self.customer.line()    # Делегирование работы внедренным экземплярам
      self.clerk.line()
      self.subject.line()
```

Часть VII, "Исключения и инструменты"

Упражнения приведены в главе 36.

1. **Оператор try/except.** Моя версия функции oops выглядит следующим образом (файл oops.py). Что касается вопросов, не связанных с написанием кода, то изменение oops для генерации исключения KeyError вместо IndexError означает, что обработчик try не перехватит исключение – оно "просочится" на верхний уровень и приведет к выдаче интерпретатором Python стандартного сообщения об ошибке. Имена KeyError и IndexError поступают из самой внешней встроенной области видимости (B в LEGB). Импортируйте модуль builtins в Python 3.X (__builtin__ в Python 2.X) и передайте его функции dir в качестве аргумента, чтобы убедиться в этом самостоятельно.

```
def oops():
   raise IndexError()

def doomed():
   try:
      oops()
   except IndexError:
      print('caught an index error!')    # перехвачена ошибка индекса!
   else:
      print('no error caught...')        # никакие ошибки
                                         #   не перехватывались...

if __name__ == '__main__': doomed()

% python oops.py
caught an index error!
```

2. **Объекты и списки исключений.** Вот как я расширил модуль собственным исключением (файл oops2.py):

```
from __future__ import print_function      # Python 2.X

class MyError(Exception): pass

def oops():
   raise MyError('Spam!')

def doomed():
   try:
      oops()
   except IndexError:
      print('caught an index error!')      # перехвачена ошибка индекса!
   except MyError as data:
      print('caught error:', MyError, data)    # перехвачена ошибка
   else:
      print('no error caught...')          # никакие ошибки
                                           #   не перехватывались...

if __name__ == '__main__':
   doomed()

% python oops2.py
caught error: <class '__main__.MyError'> Spam!
```

Как и все исключения на основе классов, экземпляр доступен через расширение as конструкции except; в сообщении об ошибке присутствует класс (<...>) и его экземпляр (Spam!). Экземпляр должен наследовать __init__ и __repr__ или __str__ из встроенного в Python класса Exception, иначе он будет выводиться во многом подобно классу. В главе 35 приводились детали о том, как это работает во встроенных классах исключений.

3. **Обработка ошибок.** Ниже демонстрируется одно из решений (файл exctools.py). Тесты находятся в файле, а не выполняются интерактивно, но результаты достаточно похожи для полной оценки. Обратите внимание, что используемый здесь подход с пустой конструкцией except и sys.exc_info будет перехватывать исключения, связанные с выходом в систему, что не делается в случае указания Exception в расширении as конструкции except; прием вряд ли можно считать идеальным в коде большинства приложений, но он полезен в инструменте подобного рода, который предназначен для работы в качестве своеобразного экрана исключений.

```
import sys, traceback

def safe(callee, *pargs, **kargs):
   try:
      callee(*pargs, **kargs)              # Перехватывать все остальное
   except:                                 # Или except Exception as E:
      traceback.print_exc()
      print('Got %s %s' % (sys.exc_info()[0], sys.exc_info()[1]))

if __name__ == '__main__':
   import oops2
   safe(oops2.oops)

c:\code> py -3 exctools.py
Traceback (most recent call last):
  File "C:\code\exctools.py", line 5, in safe
```

```
    callee(*pargs, **kargs)           # Перехватывать все остальное
  File "C:\code\oops2.py", line 6, in oops
    raise MyError('Spam!')
oops2.MyError: Spam!
Got <class 'oops2.MyError'> Spam!
```

Следующий код мог бы превратить это в декоратор функций, который обеспечит перехват исключений, генерируемых любой функцией, с применением методик, представленных в главе 32 и более подробно рассмотренных в главе 39 — он дополняет функцию, а не ожидает ее явной передачи:

```
import sys, traceback

def safe(callee):
    def callproxy(*pargs, **kargs):
        try:
            return callee(*pargs, **kargs)
        except:
            traceback.print_exc()
            print('Got %s %s' % (sys.exc_info()[0], sys.exc_info()[1]))
            raise
    return callproxy

if __name__ == '__main__':
    import oops2

    @safe
    def test():
        oops2.oops()

    test()
```

4. **Примеры для самообучения.** Ниже предлагается несколько примеров для самостоятельного изучения, если позволяет время.

```
# Поиск самого крупного файла с исходным кодом Python в отдельно взятом каталоге
import os, glob
dirname = r'C:\Python37\Lib'

allsizes = []
allpy = glob.glob(dirname + os.sep + '*.py')
for filename in allpy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])


# Поиск самого крупного файла с исходным кодом Python в целом дереве каталогов
import sys, os, pprint
if sys.platform[:3] == 'win':
    dirname = r'C:\Python37\Lib'
else:
    dirname = '/usr/lib/python'

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    for filename in filesHere:
        if filename.endswith('.py'):
```

```
            fullname = os.path.join(thisDir, filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint.pprint(allsizes[:2])
pprint.pprint(allsizes[-2:])


# Поиск самого крупного файла с исходным кодом Python
#  в пути поиска импортируемых модулей

import sys, os, pprint
visited = {}
allsizes = []

for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srcdir):
        thisDir = os.path.normpath(thisDir)
        if thisDir.upper() in visited:
            continue
        else:
            visited[thisDir.upper()] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except:
                    print('skipping', pypath)
                allsizes.append((pysize, pypath))

allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])


# Суммирование столбцов в текстовом файле с разделителями-запятыми

filename = 'data.txt'
sums = {}

for line in open(filename):
    cols = line.split(',')
    nums = [int(col) for col in cols]
    for (ix, num) in enumerate(nums):
        sums[ix] = sums.get(ix, 0) + num

for key in sorted(sums):
    print(key, '=', sums[key])


# Пример похож на предыдущий, но для хранения сумм вместо словарей
#  используются списки

import sys
filename = sys.argv[1]
numcols = int(sys.argv[2])
totals = [0] * numcols

for line in open(filename):
    cols = line.split(',')
```

```
    nums = [int(x) for x in cols]
    totals = [(x + y) for (x, y) in zip(totals, nums)]
print(totals)

# Проверка возвращения в предыдущее состояние вывода для набора сценариев
import os
testscripts = [dict(script='test1.py', args=''), # Или глобальный каталог
                                                 #   сценариев и аргументы
          dict(script='test2.py', args='spam')]
for testcase in testscripts:
    commandline = '%(script)s %(args)s' % testcase
    output = os.popen(commandline).read()
    result = testcase['script'] + '.result'
    if not os.path.exists(result):
        open(result, 'w').write(output)
        print('Created:', result)
    else:
        priorresult = open(result).read()
        if output != priorresult:
            print('FAILED:', testcase['script'])
            print(output)
        else:
            print('Passed:', testcase['script'])

# Построение графического пользовательского интерфейса
#  посредством tkinter (Tkinter в Python 2.X) с кнопками,
#  которые изменяют цвет и увеличивают размеры меток
from tkinter import *    # Использовать Tkinter в Python 2.X
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'white',
'cyan', 'purple']
def reply(text):
    print(text)
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Popup', bg='black', fg=color).pack()
    L.config(fg=color)
def timer():
    L.config(fg=random.choice(colors))
    win.after(250, timer)
def grow():
    global fontsize
    fontsize += 5
    L.config(font=('arial', fontsize, 'italic'))
    win.after(100, grow)
win = Tk()
L = Label(win, text='Spam',
          font=('arial', fontsize, 'italic'), fg='yellow', bg='navy',
          relief=RAISED)
```

```
L.pack(side=TOP, expand=YES, fill=BOTH)
Button(win, text='press', command=(lambda: reply('red'))).
pack(side=BOTTOM, fill=X)
Button(win, text='timer', command=timer).pack(side=BOTTOM, fill=X)
Button(win, text='grow', command=grow).pack(side=BOTTOM, fill=X)
win.mainloop()
```

```
# Пример похож на предыдущий, но здесь используются классы, поэтому
# каждое окно имеет собственную информацию состояния
from tkinter import *
import random

class MyGui:
    """
    Графический пользовательский интерфейс с кнопками,
    которые изменяют цвет и увеличивают размеры меток
    """
    colors = ['blue', 'green', 'orange', 'red', 'brown', 'yellow']

    def __init__(self, parent, title='popup'):
        parent.title(title)
        self.growing = False
        self.fontsize = 10
        self.lab = Label(parent, text='Gui1', fg='white', bg='navy')
        self.lab.pack(expand=YES, fill=BOTH)
        Button(parent, text='Spam', command=self.reply).pack(side=LEFT)
        Button(parent, text='Grow', command=self.grow).pack(side=LEFT)
        Button(parent, text='Stop', command=self.stop).pack(side=LEFT)

    def reply(self):
        "change the button's color at random on Spam presses"
        self.fontsize += 5
        color = random.choice(self.colors)
        self.lab.config(bg=color,
                font=('courier', self.fontsize, 'bold italic'))

    def grow(self):
        "start making the label grow on Grow presses"
        self.growing = True
        self.grower()

    def grower(self):
        if self.growing:
            self.fontsize += 5
            self.lab.config(font=('courier', self.fontsize, 'bold'))
            self.lab.after(500, self.grower)

    def stop(self):
        "stop the button growing on Stop presses"
        self.growing = False

class MySubGui(MyGui):
    colors = ['black', 'purple']  # Настройка для изменения вариантов цвета

MyGui(Tk(), 'main')
MyGui(Toplevel())
MySubGui(Toplevel())
mainloop()
```

```
# Утилита для просмотра и обслуживания входящих сообщений электронной почты
"""
Просмотр почтового ящика и извлечение только заголовков с возможностью
удалять без загрузки полного сообщения
"""

import poplib, getpass, sys

mailserver = 'your pop email server name here'    # pop.server.net
mailuser = 'your pop email user name here'
mailpasswd = getpass.getpass('Password for %s?' % mailserver)

print('Connecting...')
server = poplib.POP3(mailserver)
server.user(mailuser)
server.pass_(mailpasswd)

try:
   print(server.getwelcome())
   msgCount, mboxSize = server.stat()
   print('There are', msgCount, 'mail messages, size ', mboxSize)
   msginfo = server.list()
   print(msginfo)
   for i in range(msgCount):
      msgnum = i+1
      msgsize = msginfo[1][i].split()[1]
      resp, hdrlines, octets = server.top(msgnum, 0)    # Извлечь только
                                                        #   заголовки
      print('-'*80)
      print('[%d: octets=%d, size=%s]' % (msgnum, octets, msgsize))
      for line in hdrlines: print(line)

      if input('Print?') in ['y', 'Y']:
         for line in server.retr(msgnum)[1]: print(line) # Извлечь полное
                                                         #   сообщение
      if input('Delete?') in ['y', 'Y']:
         print('deleting')
         server.dele(msgnum)    # Удалить сообщение на сервере
      else:
         print('skipping')
finally:
   server.quit()                 # Обеспечить разблокировку почтового ящика
input('Bye.')                    # Оставить окно открытым в Windows
```

```
# Серверный сценарий CGI для взаимодействия с веб-браузером
#!/usr/bin/python
import cgi
form = cgi.FieldStorage()                    # Разбор данных формы
print("Content-type: text/html\n")           # Заголовок плюс пустая строка
print("<HTML>")
print("<title>Reply Page</title>")           # HTML-страница ответа
print("<BODY>")
if not 'user' in form:
   print("<h1>Who are you?</h1>")
else:
   print("<h1>Hello <i>%s</i>!</h1>" % cgi.escape(form['user'].value))
print("</BODY></HTML>")
```

```
# Сценарий для заполнения базы данных shelve объектами Python
# См. также примеры использования shelve в главе 28 и pickle в главе 31
rec1 = {'name': {'first': 'Bob', 'last': 'Smith'},
        'job': ['dev', 'mgr'],
        'age': 40.5}

rec2 = {'name': {'first': 'Sue', 'last': 'Jones'},
        'job': ['mgr'],
        'age': 35.0}

import shelve
db = shelve.open('dbfile')
db['bob'] = rec1
db['sue'] = rec2
db.close()


# Сценарий для вывода и обновления базы данных shelve,
#  созданной в предыдущем сценарии
import shelve
db = shelve.open('dbfile')
for key in db:
    print(key, '=>', db[key])

bob = db['bob']
bob['age'] += 1
db['bob'] = bob
db.close()


# Сценарий для заполнения и запрашивания базы данных MySql
from MySQLdb import Connect
conn = Connect(host='localhost', user='root', passwd='XXXXXXX')
curs = conn.cursor()
try:
    curs.execute('drop database testpeopledb')
except:
    pass                                        # Не существует

curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10), pay
int(4))')

curs.execute('insert people values (%s, %s, %s)', ('Bob', 'dev', 50000))
curs.execute('insert people values (%s, %s, %s)', ('Sue', 'dev', 60000))
curs.execute('insert people values (%s, %s, %s)', ('Ann', 'mgr', 40000))

curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

curs.execute('select * from people where name = %s', ('Bob',))
print(curs.description)
colnames = [desc[0] for desc in curs.description]
while True:
    print('-' * 30)
    row = curs.fetchone()
    if not row: break
```

```
    for (name, value) in zip(colnames, row):
        print('%s => %s' % (name, value))
conn.commit()                              # Сохранить вставленные записи
```

```
# Извлечение и открытие/воспроизведение файла по протоколу FTP
import webbrowser, sys
from ftplib import FTP                     # Инструменты FTP на основе сокетов
from getpass import getpass                # Скрытый ввод пароля
if sys.version[0] == '2': input = raw_input    # Совместимость с Python 2.X

nonpassive = False            # Использовать активный режим FTP для сервера?
filename = input('File?')     # Загружаемый файл
dirname = input('Dir? ') or '.'   # Удаленный каталог
sitename = input('Site?')     # Сайт FTP
user = input('User?')         # Использовать () для анонимного доступа
if not user:
    userinfo = ()
else:
    from getpass import getpass    # Скрытый ввод пароля
    userinfo = (user, getpass('Pswd?'))

print('Connecting...')
connection = FTP(sitename)    # Подключение к сайту FTP
connection.login(*userinfo)   # По умолчанию анонимный вход
connection.cwd(dirname)       # Передавать по 1 Кбайт за раз в локальный файл
if nonpassive:              # Применить активный режим FTP, если сервер требует
    connection.set_pasv(False)

print('Downloading...')
localfile = open(filename, 'wb')       # Локальный файл для хранения
                                       #   загруженных данных
connection.retrbinary('RETR ' + filename, localfile.write, 1024)
connection.quit()
localfile.close()

print('Playing...')
webbrowser.open(filename)
```

Предметный указатель

A

ASCII, 414

B

BOM (Byte order mark), 409

C

Cygwin, 655

D

DFLR (Depth-First, Left-to-Right), 192

M

MRO (Method Resolution Order), 234

P

Python 2.3, 2.4, 2.5, 691
Python 2.6, 687
Python 2.7, 683
Python 3.0, 687; 688
Python 3.1, 686
Python 3.2, 686
Python 3.3, 685
Python 3.7, 683
Python 3.8, 683

U

Unicode, 400; 406; 414; 440
Unix, 668

W

Windows, 669

X

XML, 447

А

Алгоритм
  MRO, 234; 238
  наследования Python, 618; 620
  сопоставления, 573
Аннотации функций, 576
Аргумент
  декоратора, 516; 552; 576
  добавление аргументов к декоратору, 533
  ключевой, 519
  командной строки интерпретатора
    Python, 663
  метода дескриптора, 463
  порядок передачи аргументов, 572
  произвольный, 574
  функции, 566
Атрибут
  вычисляемый, 459; 467
  дескрипторы атрибутов
    только для чтения, 464
    уровня класса, 257
  закрытый, 549
  индивидуальный, 63
  объявление атрибутов, 247
  открытый, 549
  перехват атрибутов
    встроенных в Python 3.X, 76
    для встроенных операций, 484
  псевдозакрытый, 83; 492
  средства доступа к атрибутам, 456
  управляемый, 455
  функции, 522
  экземпляра, 81; 510
    класса, 519; 520

Б

База данных, 92
Библиотека, 681
  стандартная, 681

В

Вывод
  стандартный, 367
Вызов
  обратный, 160

Д

Данные экземпляра, 251
Декодирование, 403; 416
  при файловом вводе, 436
Декоратор, 271; 504; 506; 507; 531; 543
  аргументы декоратора, 516; 552
  вложение декораторов, 514; 575
  для установки и удаления, 461
  классов, 271; 275; 505; 536; 539; 544; 639
  отслеживание интерфейсов с помощью
    декораторов классов, 539
  применение декораторов классов к
    встроенным типам, 541

компромиссы, связанные с декораторами, 556
применение к методам, 637
трассировка с помощью метаклассов и декораторов, 636
функции, 161; 179; 271; 504; 508; 518; 544; 565
Декорирование, 272; 504
вручную, 635
методов, 524
использование вложенных функций для декорирования методов, 526
использование дескрипторов для декорирования методов, 527
функций, 270
Делегирование, 178; 225; 271; 538; 551
Дескриптор, 246; 251; 252; 259; 462
аргументы методов дескриптора, 463
атрибутов только для чтения, 464
данных, 619
для проверки достоверности, 494
Деструктор __del__, 166
Диспетчер контекстов, 349

З

Запись, 49
Запускающий модуль Windows, 670

И

Иерархия
классов, 38
настраиваемая, 71
Индексирование, 127
Инкапсуляция, 61
Инструмент, 681
интроспекции Python, 78
Интерпретатор Python
аргументы командной строки, 663
конфигурирование, 658
установка, 655
Интерфейс
объектный, 506; 538
отслеживание интерфейсов с помощью декораторов классов, 539
Исключение, 316; 357
SystemExit, 388
генерация исключений, 342
иерархия исключений, 362
категории встроенных исключений, 366
классы исключений
встроенные, 365
на основе классов, 359
обработка исключений, 327
обработчик исключений, 317; 375
перехват исключений, 330
сцепление исключений в Python 3.X, 345
Итератор
активный, 140

К

Класс, 21; 24; 54; 229
ArithmeticError, 365
BaseException, 365
Exception, 360; 365
ListInstance, 195
ListTree, 254
LookupError, 366
Manager, 54
object, 232
Person, 54
Spam, 304
type, 232
абстрактный суперкласс, 106
декораторы классов, 275; 539; 544
иерархия классов, 38
изменение во время выполнения, 286
классический, 235
метакласс, 221; 512; 615
методы класса, 266
множество классов, 543
написание классов, 34
настройка через наследование, 38
нового стиля, 235; 284
-оболочка, 556
-одиночка, 536
перенаправляющий, 561
переносимость классов, 237
подкласс, 38
подмешиваемый, 193; 281; 488
-посредник, 225; 227
проектирование с использованием классов, 169
расширенные возможности классов, 216
реализация классов, 96
связывание классов, 104
создание, 46
суперкласс, 488; 559; 615
подмешиваемый, 559

управление классами, 630
напрямую, 547
экземпляр класса, 232
Код
сопровождение кода, 67
Кодирование, 403; 416
при файловом выводе, 436
символов, 402
строк Unicode в Python 2.X, 420
Композиция, 21
Конструктор, 43
__init__, 166
метод конструктора, 43
написание кода, 55
настройка конструкторов, 72
суперкласса, 101
Конфигурирование интерпретатора Python, 658
Кортеж
именованный, 51

Л

Литерал
строковый, 410

М

Макрос, 507; 508
Метакласс, 221; 231; 260; 270; 271; 276; 512; 615; 639
методы метаклассов, 270; 623
трассировка с помощью метаклассов, 636
Метафункция, 272; 508
Метод, 99
__add__, 157
__bool__, 163
__call__, 158; 162
__cmp__, 163
__dict__, 194
__getattr__, 480; 498
__getattribute__, 480; 500
__gt__, 162
__iadd__, 157
__len__, 163
__lt__, 162
__nonzero__, 164
булевский, 164
вызов методов, 425
вызовы методов, 25
декорирование методов, 524
использование вложенных функций для декорирования методов, 526
использование дескрипторов для декорирования методов, 527
класса, 102; 262; 266; 268
конструктора, 43
метакласса, 623; 267
перегрузка операций в методах метакласса, 624
написание кода, 61
несвязанный, 183
перегрузки операций, 562
расширение методов, 66
связанный, 160; 183
стандартный, 238
статический, 261; 262; 266
экземпляра, 101; 262; 266
Множество
классов, 543
экземпляров, 543
Модуль, 120
doctest, 392
pickle, 446
struct, 444
собирающий, 211

Н

Наборы символов, 402
Нарезание, 127
Наследование, 20; 102; 171; 551; 617
множественное, 192; 303
настройка через наследование, 65
присваивания, 621
ромбовидное, 234
Настройка через наследование, 65

О

Обработчик исключений, 317; 375
Объект, 508
итерируемый, 532
определяемый пользователем, 132
класса, 34
-посредник, 633
пространства имен, 57
фабрика объектов, 57
экземпляра, 34; 342

Оператор
assert, 347
class, 96
raise, 341; 344; 371; 379
try, 328; 338; 383
try/except/else, 327
try/except/finally, 337
try/finally, 335
Операция
методы перегрузки операций, 562
над последовательностями, 426
перегрузка операций, 42; 45; 123; 283
Отладчик, 393
Ошибки связанные с классами, 524

П

Перегрузка операций, 22; 42; 45; 63; 123; 283
в методах метакласса, 624
Переменная
нелокальная, 521
Перехват исключений, 330
Перехват срезов, 127
Подкласс, 38
написание кода, 66
Полиморфизм, 30; 69
Последовательность
операции над последовательностями, 426
Посредники
вызовов, 505
интерфейсов, 505
Предупреждения, 381
Программирование
аспектно-ориентированное, 507
объектно-ориентированное, 20; 71; 169; 648
путем настройки, 31
функциональное, 35
Проектирование
с использованием классов, 169
Пространство имен, 108
словари пространств имен, 248
Протокол координирования, 633
Профилировщик, 393

Р

Расширения, доступные только в Python 3.X, 679
Рефакторинг, 62

С

Свойства, 246; 251; 256; 457
реализация свойств с помощью декораторов, 460
Словари пространств имен, 248
Слот, 246; 247
правила использования слотов, 252
Срез
перехват срезов, 127
Строка, 402
Unicode, 414
кодирование символов, 402
кодирование строк Unicode в Python 2.X, 420
преобразования строковых типов, 412
типы строк Python, 406
хранение строк Python в памяти, 405
Строковые литералы Python 3.X, 410
Структура, 65
Суперкласс, 234; 268; 488; 559; 615
object, 234
подмешиваемый, 559

Т

Текст
ASCII, 414
декодирование, 416
кодирование, 416
отличный от ASCII, 415
Тестирование, 532
Тип, 229
bytearray, 428
bytes, 425
str, 425
встроенный, 541
применение декораторов классов к встроенным типам, 541
проверка типов, 578
расширение встроенных типов, 217
путем внедрения, 217
путем создания подклассов, 218
строковый, 407
преобразования строковых типов, 412
Трассировка с помощью метаклассов и декораторов, 636

У

Установка интерпретатора Python, 654

Ф

Фабрика
  записей, 59
  объектов, 57; 190
Файл
  Unicode, 440
  двоичный, 408; 431
  текстовый, 408; 431
Фреймворк, 32
Функция
  isinstance, 232
  super, 277
  testdriver, 384
  аннотации функций, 576
  аргументы функций, 566
  декорирование функций, 270
  замыкания, 520; 521
  метафункция, 272; 508
  управление функциями напрямую, 547
  управляющая, 543

Э

Экземпляр, 24; 50; 54; 262; 264; 514
  атрибут экземпляра, 510; 520
  данные экземпляра, 251
  класса, 232
  множество экземпляров, 543
  управление экземплярами, 630

Я

Язык
  XML, 447

ГЛУБОКОЕ ОБУЧЕНИЕ
ГОТОВЫЕ РЕШЕНИЯ

Давид Осинга

www.williamspublishing.com

Благодаря готовым примерам, приведенным в книге, вы научитесь решать задачи, связанные с классификацией и генерированием текста, изображений и музыки. В каждой главе описывается несколько решений, объединяемых в единый проект, например приложение, реализующее тренировку музыкальной рекомендательной системы. Также имеется глава с описанием методик, которые в случае необходимости помогут выполнить отладку нейронной сети. Основные темы книги:

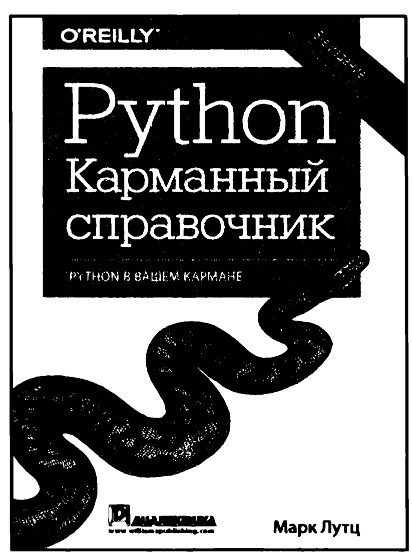
- использование векторных представлений слов для вычисления схожести текстов;
- построение рекомендательной системы фильмов на основе ссылок в Википедии;
- визуализация внутренних состояний нейронной сети;
- создание модели, рекомендующей эмодзи для фрагментов текста;
- повторное использование предварительно обученных сетей для создания службы обратного поиска изображений;
- генерирование пиктограмм с помощью генеративно-состязательных сетей (GAN), автокодировщиков и рекуррентных сетей (RNN);
- распознавание музыкальных жанров и индексирование коллекций песен.

ISBN: 978-5-907144-50-7

в продаже

PYTHON КАРМАННЫЙ СПРАВОЧНИК
ПЯТОЕ ИЗДАНИЕ

Марк Лутц



www.dialektika.com

Этот краткий справочник по Python составлен с учетом версий 3.4 и 2.7 и очень удобен для наведения быстрых справок при разработке программ на Python. В лаконичной форме здесь представлены все необходимые сведения о типах данных и операторах Python, специальных методах перегрузки операторов, встроенных функциях и исключениях, наиболее употребительных стандартных библиотечных модулях и других примечательных языковых средствах Python, в том числе и для объектно-ориентированного программирования. Справочник рассчитан на широкий круг читателей, интересующихся программированием на Python.

ISBN 978-5-907114-60-9 **в продаже**

СТАНДАРТНАЯ БИБЛИОТЕКА PYTHON 3
СПРАВОЧНИК С ПРИМЕРАМИ
2-Е ИЗДАНИЕ

Даг Хеллман

www.dialektika.com

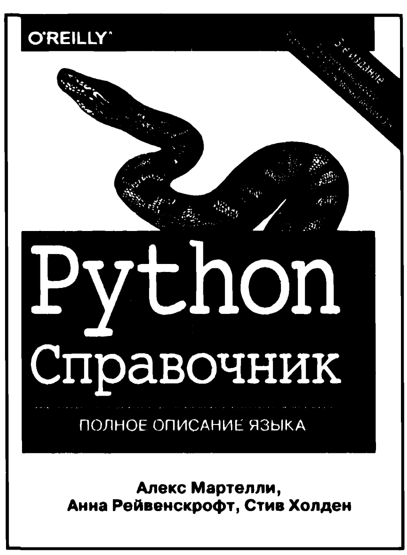
В этой книге Даг Хеллман, эксперт по языку Python, описывает все основные разделы библиотеки Python 3.x, сопровождая изложение материала компактными примерами исходного кода и результатами их выполнения. Приведенные примеры наглядно демонстрируют возможности всех модулей, предлагаемых библиотекой. Каждому модулю посвящен отдельный раздел, содержащий ссылки на дополнительные ресурсы, что делает эту книгу идеальным учебным и справочным руководством. Основные темы книги:

- манипулирование текстом с помощью модулей `string`, `textwrap`, `re` (регулярные выражения) и `difflib`;
- использование структур данных: модули `enum`, `collections`, `array`, `heapq`, `queue`, `struct`, `copy` и множество других;
- элегантная и компактная реализация алгоритмов с использованием модулей `functools`, `itertools` и `contextlib`;
- обработка значений даты и времени и решение сложных математических задач;
- архивирование и сжатие данных.

ISBN: 978-5-6040043-8-8 **в продаже**

PYTHON
СПРАВОЧНИК
ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА
3-Е ИЗДАНИЕ

Алекс Мартелли
Анна Рейвенскрофт
Стив Холден



www.williamspublishing.com

Книга охватывает чрезвычайно широкий спектр областей применения Python, включая веб-приложения, сетевое программирование, обработку XML-документов, взаимодействие с базами данных и высокоскоростные вычисления. Она станет идеальным подспорьем как для тех, кто решил изучить Python, имея предварительный опыт программирования на других языках, так и для тех, кто уже использует этот язык в своих разработках. Основные темы книги:

- синтаксис Python, модули стандартной библиотеки и пакеты расширений;
- операции с файлами, работа с текстом, базы данных, многозадачность и обработка числовых данных;
- основы работы с сетями, и клиентские модули сетевых протоколов;
- модули расширения Python, средства пакетирования и распространения расширений, модулей и приложений.

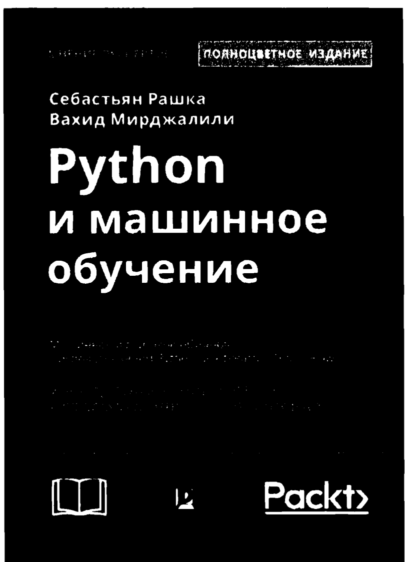
ISBN 978-5-6040723-8-7

в продаже

PYTHON И МАШИННОЕ ОБУЧЕНИЕ

МАШИННОЕ И ГЛУБОКОЕ ОБУЧЕНИЕ С ИСПОЛЬЗОВАНИЕМ PYTHON, SCIKIT-LEARN И TENSORFLOW, 2-Е ИЗДАНИЕ

Себастьян Рашка и Вахид Мирджалили



www.dialektika.com

Машинное обучение поглощает мир программного обеспечения, и теперь глубокое обучение расширяет машинное обучение. Освойте и работайте с передовыми технологиями машинного обучения, нейронных сетей и глубокого обучения с помощью 2-го издания бестселлера Себастьяна Рашки. Будучи основательно обновленной с учетом самых последних библиотек Python с открытым кодом, эта книга предлагает практические знания и приемы, которые необходимы для создания и содействия машинному обучению, глубокому обучению и современному анализу данных. Если вы читали 1-е издание книги, то вам доставит удовольствие найти новый баланс классических идей и современных знаний в машинном обучении. Каждая глава была серьезно обновлена, и появились новые главы по ключевым технологиям. У вас будет возможность изучить и поработать с TensorFlow более вдумчиво, нежели ранее, а также получить важнейший охват библиотеки для нейронных сетей Keras наряду с самыми свежими обновлениями библиотеки scikit-learn.

ISBN 978-5-907114-52-4 в продаже

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ С ПРИМЕРАМИ НА PYTHON

Пратик Джоши

www.dialektika.com

В этой книге исследуются различные сценарии применения искусственного интеллекта. Вначале рассматриваются общие концепции искусственного интеллекта, после чего обсуждаются более сложные темы, такие как предельно случайные леса, скрытые марковские модели, генетические алгоритмы, сверточные нейронные сети и др. Вы узнаете о том, как принимать обоснованные решения при выборе необходимых алгоритмов, а также о том, как реализовывать эти алгоритмы на языке Python для достижения наилучших результатов.

Основные темы книги:

- различные методы классификации и регрессии данных;
- создание интеллектуальных рекомендательных систем;
- логическое программирование и способы его применения;
- построение автоматизированных систем распознавания речи;
- основы эвристического поиска и генетического программирования;
- разработка игр с использованием искусственного интеллекта;
- обучение с подкреплением;
- алгоритмы глубокого обучения и создание приложений на их основе.

ISBN: 978-5-907114-41-8 **в продаже**

АВТОМАТИЗАЦИЯ РУТИННЫХ ЗАДАЧ С ПОМОЩЬЮ PYTHON

практическое руководство для начинающих

Эл Свейгарт

www.williamspublishing.com

Книга научит вас использовать Python для написания программ, способных в считанные секунды сделать то, на что раньше у вас уходили часы ручного труда, причем никакого опыта программирования от вас не требуется. Как только вы овладеете основами программирования, вы сможете создавать программы на языке Python, которые будут без труда выполнять в автоматическом режиме различные полезные задачи, такие как:

- поиск определенного текста в файле или в множестве файлов;
- создание, обновление, перемещение и переименование файлов и папок;
- поиск в Интернете и загрузка онлайн-контента;
- обновление и форматирование данных в электронных таблицах Excel любого размера;
- разбиение, слияние, разметка водяными знаками и шифрование PDF-документов;
- рассылка напоминаний в виде сообщений электронной почты или текстовых уведомлений;
- заполнение онлайновых форм.

ISBN 978-5-6040724-2-4 **в продаже**

Изучаем Python, том 2

С помощью этой практической книги вы получите всестороннее и глубокое введение в основы языка Python. Будучи основанным на популярном учебном курсе Марка Лутца, обновленное 5-е издание книги поможет вам быстро научиться писать эффективный высококачественный код на Python. Она является идеальным способом начать изучение Python, будь вы новичок в программировании или профессиональный разработчик программного обеспечения на других языках.

Это простое и понятное учебное пособие, укомплектованное контрольными вопросами, упражнениями и полезными иллюстрациями, позволит вам освоить основы линеек Python 3.X и 2.X. Вы также ознакомитесь с расширенными возможностями языка, получившими широкое распространение в коде Python.

“*Книга* Learning Python *находится в начале моего списка рекомендованной литературы для любого, кто желает научиться программировать на Python.*”

Дуг Хеллманн
старший инженер-программист, Racemi, Inc.

Благодаря книге вы:

- Исследуете основные встроенные типы объектов Python, такие как числа, списки и словари
- Научитесь создавать и обрабатывать объекты с помощью операторов Python и освоите общую синтаксическую модель Python
- Сможете применять функции для устранения избыточности кода и упаковки кода с целью многократного использования
- Узнаете, как организовывать операторы, функции и прочие инструменты в более крупные компоненты посредством модулей
- Погрузитесь глубже в классы — инструмент объектно-ориентированного программирования Python для структурирования кода
- Научитесь писать крупные программы с применением модели обработки исключений и инструментов разработки Python
- Освоите более сложные инструменты Python, включая декораторы, дескрипторы, метаклассы и обработку Unicode

Марк Лутц является мировым лидером в обучении языку Python, автором самых ранних и ставших бестселлерами книг по Python, а также первопроходцем в сообществе Python, начиная с 1992 года. Обладая более чем 30-летним опытом разработки, Марк был автором книг *Programming Python, 4th Edition* издательства O’Reilly и *Python. Карманный справочник, 5-е издание* (пер. с англ., издательство “Диалектика”, 2014 г.)

Категория: программирование
Предмет рассмотрения: Python
Уровень: для начинающих и пользователей средней квалификации

ISBN 978-5-907144-53-8

www.williamspublishing.com

oreilly.com