

Авторитетный курс объектно-ориентированного программирования

5-е издание

Изучаем

Python



Том 1

DАТАЛЕКСИКА
O'REILLY®

Марк Лутц

5-е издание

Изучаем Python

FIFTH EDITION

Learning Python

Mark Lutz

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

5-е издание

Изучаем Python

Том 1

Марк Лутц



Москва · Санкт-Петербург
2019

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 1, 5-е изд. : Пер. с англ. – СПб. : ООО “Диалектика”, 2019. – 832 с. : ил. – Парал. тит. англ.

ISBN 978-5-907144-52-1 (рус., том 1)

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 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 1

5-е издание

Подписано в печать 05.06.2019. Формат 70×100/16.

Гарнитура Times.

Усл. печ. л. 55. Уч.-изд. л. 67,08.

Тираж 500 экз. Заказ № 5983.

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 978-5-907144-52-1 (рус., том 1)

ISBN 978-5-907144-51-4 (рус., многотом.)

ISBN 978-1-449-35573-9 (англ.)

© 2019, ООО “Диалектика”

© 2013 Mark Lutz

Оглавление

Часть I. Начало работы	39
ГЛАВА 1. Python в вопросах и ответах	40
ГЛАВА 2. Как Python выполняет программы	64
ГЛАВА 3. Как пользователь выполняет программы	78
Часть II. Типы и операции	125
ГЛАВА 4. Введение в типы объектов Python	126
ГЛАВА 5. Числовые типы	165
ГЛАВА 6. Кратко о динамической типизации	207
ГЛАВА 7. Фундаментальные основы строк	219
ГЛАВА 8. Списки и словари	268
ГЛАВА 9. Кортежи, файлы и все остальное	302
Часть III. Операторы и синтаксис	343
ГЛАВА 10. Введение в операторы Python	344
ГЛАВА 11. Операторы присваивания, выражений и вывода	362
ГЛАВА 12. Проверки <code>if</code> и правила синтаксиса	394
ГЛАВА 13. Циклы <code>while</code> и <code>for</code>	410
ГЛАВА 14. Итерации и включения	438
ГЛАВА 15. Документация	466
Часть IV. Функции и генераторы	491
ГЛАВА 16. Основы функций	492
ГЛАВА 17. Области видимости	504
ГЛАВА 18. Аргументы	541
ГЛАВА 19. Расширенные возможности функций	571
ГЛАВА 20. Включения и генераторы	598
ГЛАВА 21. Оценочные испытания	645
Часть V. Модули и пакеты	683
ГЛАВА 22. Модули: общая картина	684
ГЛАВА 23. Основы написания модулей	702
ГЛАВА 24. Пакеты модулей	720
ГЛАВА 25. Расширенные возможности модулей	758
ПРИЛОЖЕНИЕ. Решения упражнений, приводимых в конце частей	794
Предметный указатель	819

Содержание

Об авторе	21
Об иллюстрации на обложке	21
Предисловие	22
“Экосистема” этой книги	22
О пятом издании	23
Линейки Python 2.X и Python 3.X	25
Современная история Python 2.X/3.X	25
Раскрытие линеек Python 3.X и Python 2.X	26
Какая версия Python должна использоваться?	27
Предпосылки и усилия	28
Структура этой книги	29
Чем эта книга не является	32
Это не справочник и не руководство по специфическим приложениям	32
Это не краткая история для спешащих людей	33
Изложение последовательно до той степени, до которой позволяет Python	33
Программы в книге	34
Версии Python	34
Платформы	35
Загрузка кода примеров для книги	35
Использование кода, сопровождающего книгу	35
Соглашения, используемые в этой книге	36
Ждем ваших отзывов!	36
Благодарности	37
Предыстория	37
Благодарности Python	37
Личные благодарности	38
Часть I. Начало работы	39
ГЛАВА 1. Python в вопросах и ответах	40
Почему люди используют Python?	40
Качество программного обеспечения	42
Продуктивность труда разработчиков	42
Является ли Python “языком написания сценариев”?	43
Хорошо, но в чем недостаток?	44
Кто использует Python в наши дни?	46
Что можно делать с помощью Python?	48
Системное программирование	48
Графические пользовательские интерфейсы	49
Написание сценариев для Интернета	49
Интеграция компонентов	50
Программирование для баз данных	50
Быстрое прототипирование	51
Численное и научное программирование	51

И еще: игры, изображения, глубинный анализ данных, роботы, электронные таблицы Excel...	52
Как Python разрабатывается и поддерживается?	53
Компромиссы, связанные с открытым кодом	53
Каковы технические превосходства Python?	54
Он объектно-ориентированный и функциональный	54
Он бесплатный	55
Он переносимый	55
Он мощный	56
Он смешиваемый	57
Он относительно прост в использовании	57
Он относительно прост в изучении	58
Он назван в честь группы “Монти Пайтон”	58
Как Python соотносится с языком X?	59
Резюме	60
Проверьте свои знания: контрольные вопросы	61
Проверьте свои знания: ответы	61
ГЛАВА 2. Как Python выполняет программы	64
Введение в интерпретатор Python	64
Выполнение программ	66
Точка зрения программиста	66
Точка зрения Python	67
Разновидности модели выполнения	70
Альтернативные реализации Python	70
Инструменты оптимизации выполнения	73
Фиксированные двоичные файлы	75
Будущие возможности?	76
Резюме	76
Проверьте свои знания: контрольные вопросы	77
Проверьте свои знания: ответы	77
ГЛАВА 3. Как пользователь выполняет программы	78
Интерактивная подсказка	78
Запуск интерактивного сеанса	79
Пути поиска в системе	81
Новые возможности для Windows в версии Python 3.3:	
переменная среды PATH и запускающий модуль	81
Где выполнять: каталоги для кода	82
Что не набирать: приглашения к вводу и комментарии	83
Интерактивное выполнение кода	84
Для чего нужна интерактивная подсказка?	85
Замечания по использованию: интерактивная подсказка	87
Командная строка системы и файлы	89
Первый сценарий	90
Запуск файлов в командной строке	91
Варианты использования командной строки	92
Замечания по использованию: командная строка и файлы	93

Исполняемые сценарии в стиле Unix: #!	94
Основы сценариев Unix	94
Трюк с поиском посредством env в Unix	95
Запускающий модуль для Windows в версии Python 3.3: #! приходит в Windows	95
Щелчки на значках файлов	97
Основы щелчков на значках	97
Щелчки на значках в Windows	98
Трюк с использованием функции input в Windows	99
Другие ограничения, связанные со щелчками на значках	101
Импортирование и повторная загрузка модулей	101
Основы импортирования и повторной загрузки	101
Дополнительная история о модулях: атрибуты	103
Замечания по использованию: import и reload	106
Использование exec для выполнения файлов модулей	107
Пользовательский интерфейс IDLE	108
Детали запуска IDLE	109
Базовое использование IDLE	110
Удобные функциональные возможности IDLE	111
Расширенные инструменты IDLE	112
Замечания по использованию: IDLE	112
Другие IDE-среды	114
Другие варианты запуска	116
Встраивание вызовов	116
Фиксированные двоичные исполняемые файлы	117
Варианты запуска из текстовых редакторов	117
Прочие варианты запуска	117
Будущие возможности?	118
Какой вариант должен использоваться?	118
Резюме	120
Проверьте свои знания: контрольные вопросы	120
Проверьте свои знания: ответы	121
Проверьте свои знания: упражнения для части I	122
Часть II. Типы и операции	125
ГЛАВА 4. Введение в типы объектов Python	126
Концептуальная иерархия Python	126
Для чего используются встроенные типы?	127
Основные типы данных Python	128
Числа	130
Строки	131
Операции над последовательностями	132
Неизменяемость	134
Методы, специфичные для типа	135
Получение справки	136
Другие способы написания строк	137
Строки Unicode	138
Сопоставление с образцом	141

Списки	141
Операции над последовательностями	141
Операции, специфичные для типа	142
Контроль границ	142
Вложение	143
Списковые включения	144
Словари	146
Операции над отображениями	146
Снова о вложении	147
Недостающие ключи: проверки <code>if</code>	149
Сортировка ключей: циклы <code>for</code>	150
Итерация и оптимизация	152
Кортежи	153
Для чего используются кортежи?	154
Файлы	154
Файлы с двоичными байтами	155
Файлы с текстом Unicode	156
Другие инструменты, подобные файлам	158
Прочие основные типы	158
Как нарушить гибкость кода	160
Классы, определяемые пользователем	161
Все остальное	162
Резюме	162
Проверьте свои знания: контрольные вопросы	163
Проверьте свои знания: ответы	163
ГЛАВА 5. Числовые типы	165
Основы числовых типов	165
Числовые литералы	166
Встроенные инструменты для обработки объектов чисел	168
Операции выражений Python	168
Числа в действии	173
Переменные и базовые выражения	173
Форматы числового отображения	175
Сравнения: нормальные и сцепленные	176
Деление: классическое, с округлением в меньшую сторону и настоящее	178
Точность целых чисел	182
Комплексные числа	183
Шестнадцатеричная, восьмеричная и двоичная формы записи: литералы и преобразования	183
Побитовые операции	185
Другие встроенные инструменты для обработки чисел	187
Другие числовые типы	189
Десятичные типы	189
Дробный тип	192
Множества	195
Булевские значения	203
Численные расширения	204
Резюме	205

Проверьте свои знания: контрольные вопросы	205
Проверьте свои знания: ответы	205
ГЛАВА 6. Кратко о динамической типизации	207
Случай отсутствия операторов объявления	207
Переменные, объекты и ссылки	208
Типы обитают в объектах, не в переменных	210
Объекты подвергаются сборке мусора	210
Разделяемые ссылки	212
Разделяемые ссылки и изменения на месте	214
Разделяемые ссылки и равенство	215
Динамическая типизация вездесуща	217
Резюме	218
Проверьте свои знания: контрольные вопросы	218
Проверьте свои знания: ответы	218
ГЛАВА 7. Фундаментальные основы строк	219
Вопросы, раскрываемые в главе	219
Unicode: краткая история	220
Основы строк	220
Строковые литералы	222
Строки в одинарных и двойных кавычках являются одинаковыми	223
Управляющие последовательности представляют специальные символы	223
Неформатированные строки подавляют управляющие последовательности	227
Утроенные кавычки представляют многострочные блочные строки	228
Строки в действии	230
Базовые операции	230
Индексация и нарезание	231
Инструменты преобразования строк	235
Изменение строк, часть I	238
Строковые методы	239
Синтаксис вызова методов	239
Методы строк	240
Примеры строковых методов: изменение строк, часть II	241
Примеры строковых методов: разбор текста	243
Другие распространенные строковые методы в действии	244
Функции первоначального модуля <code>string</code> (изъяты из Python 3.X)	245
Выражения форматирования строк	246
Основы выражений форматирования	247
Расширенный синтаксис выражений форматирования	248
Более сложные примеры использования выражений форматирования	249
Выражения форматирования, основанные на словаре	250
Вызовы методов форматирования строк	251
Основы методов форматирования	251
Добавление ключей, атрибутов и смещений	252
Расширенный синтаксис методов форматирования	253
Более сложные примеры использования методов форматирования	254
Сравнение с выражением форматирования %	256
Для чего используется метод <code>format?</code>	259

Общие категории типов	264
Типы разделяют наборы операций по категориям	264
Изменяемые типы можно модифицировать на месте	265
Резюме	266
Проверьте свои знания: контрольные вопросы	266
Проверьте свои знания: ответы	267
ГЛАВА 8. Списки и словари	268
Списки	268
Списки в действии	270
Базовые списковые операции	271
Итерация по спискам и списковые включения	271
Индексация, нарязание и матрицы	272
Изменение списков на месте	273
Словари	279
Словари в действии	280
Базовые словарные операции	281
Изменение словарей на месте	282
Дополнительные словарные методы	283
Пример: база данных о фильмах	285
Замечания по использованию словарей	287
Другие способы создания словарей	291
Изменения в словарях в Python 3.X и 2.7	293
Резюме	300
Проверьте свои знания: контрольные вопросы	301
Проверьте свои знания: ответы	301
ГЛАВА 9. Кортежи, файлы и все остальное	302
Кортежи	303
Кортежи в действии	304
Для чего используются списки и кортежи?	307
Снова о записях: именованные кортежи	307
Файлы	309
Открытие файлов	310
Использование файлов	311
Файлы в действии	313
Кратко о текстовых и двоичных файлах	314
Хранение объектов Python в файлах: преобразования	315
Хранение собственных объектов Python: модуль pickle	317
Хранение объектов Python в формате JSON	318
Хранение упакованных двоичных данных: модуль struct	320
Диспетчеры контекстов для файлов	321
Другие файловые операции	322
Обзор и сводка по основным типам	323
Гибкость объектов	324
Ссылки или копии	326
Сравнения, равенство и истинность	328
Смысл понятий “истина” и “ложь” в Python	331
Иерархии типов Python	333

Объекты <code>type</code>	333
Прочие типы в Python	335
Затруднения, связанные со встроенными типами	335
Присваивание создает ссылки, а не копии	336
Повторение добавляет один уровень глубины	336
Остерегайтесь циклических структур данных	337
Неизменяемые типы нельзя модифицировать на месте	338
Резюме	338
Проверьте свои знания: контрольные вопросы	339
Проверьте свои знания: ответы	339
Проверьте свои знания: упражнения для части II	340
Часть III. Операторы и синтаксис	343
ГЛАВА 10. Введение в операторы Python	344
Еще раз о концептуальной иерархии Python	344
Операторы Python	345
История о двух <code>if</code>	347
Что Python добавляет	347
Что Python устранил	348
Для чего используется синтаксис с отступами?	349
Несколько специальных случаев	352
Короткий пример: интерактивные циклы	354
Простой интерактивный пример	354
Выполнение математических действий над пользовательским вводом	356
Обработка ошибок путем проверки ввода	357
Обработка ошибок с помощью оператора <code>try</code>	358
Вложение кода на три уровня в глубину	360
Резюме	361
Проверьте свои знания: контрольные вопросы	361
Проверьте свои знания: ответы	361
ГЛАВА 11. Операторы присваивания, выражений и вывода	362
Операторы присваивания	362
Формы оператора присваивания	363
Присваивание последовательности	364
Расширенная распаковка последовательностей в Python 3.X	367
Групповые присваивания	371
Дополненные присваивания	372
Правила именования переменных	375
Операторы выражений	379
Операторы выражений и изменения на месте	380
Операции вывода	380
Функция <code>print</code> в Python 3.X	381
Оператор <code>print</code> в Python 2.X	384
Перенаправление потока вывода	385
Вывод, нейтральный к версии	389
Резюме	392

Проверьте свои знания: контрольные вопросы	393
Проверьте свои знания: ответы	393
ГЛАВА 12. Проверки <code>if</code> и правила синтаксиса	394
Операторы <code>if</code>	394
Общий формат	394
Элементарные примеры	395
Множественное ветвление	395
Снова о синтаксисе Python	398
Ограничители блоков: правила отступов	399
Ограничители операторов: строки и продолжения	401
Несколько особых случаев	402
Значения истинности и булевские проверки	404
Тернарное выражение <code>if/else</code>	405
Резюме	408
Проверьте свои знания: контрольные вопросы	409
Проверьте свои знания: ответы	409
ГЛАВА 13. Циклы <code>while</code> и <code>for</code>	410
Циклы <code>while</code>	410
Общий формат	410
Примеры	411
Операторы <code>break</code> , <code>continue</code> , <code>pass</code> и конструкция <code>else</code> цикла	412
Общий формат цикла	412
Оператор <code>pass</code>	412
Оператор <code>continue</code>	414
Оператор <code>break</code>	414
Конструкция <code>else</code> цикла	415
Циклы <code>for</code>	417
Общий формат	418
Примеры	418
Методики написания циклов	425
Циклы с подсчетом: <code>range</code>	425
Просмотр последовательностей: <code>while</code> и <code>range</code> или <code>for</code>	426
Тасование последовательностей: <code>range</code> и <code>len</code>	427
Неполный обход: <code>range</code> или срезы	428
Изменение списков: <code>range</code> или включения	429
Параллельные обходы: <code>zip</code> и <code>map</code>	430
Генерация смещений и элементов: <code>enumerate</code>	433
Резюме	436
Проверьте свои знания: контрольные вопросы	437
Проверьте свои знания: ответы	437
ГЛАВА 14. Итерации и включения	438
Итерации: первый взгляд	438
Протокол итерации: итераторы файловых объектов	439
Ручная итерация: <code>iter</code> и <code>next</code>	442
Итерируемые объекты других встроенных типов	445

Списковые включения: первый подробный взгляд	447
Основы списковых включений	448
Использование списковых включений с файлами	449
Расширенный синтаксис списковых включений	451
Другие итерационные контексты	453
Новые итерируемые объекты в Python 3.X	457
Влияние на код Python 2.X: доводы за и против	457
Итерируемый объект <code>range</code>	458
Итерируемые объекты <code>map</code> , <code>zip</code> и <code>filter</code>	459
Итераторы с множеством проходов или с одним проходом	461
Итерируемые словарные представления	462
Другие темы, связанные с итерацией	464
Резюме	464
Проверьте свои знания: контрольные вопросы	465
Проверьте свои знания: ответы	465
ГЛАВА 15. Документация	466
Источники документации Python	466
Комментарии <code>#</code>	467
Функция <code>dir</code>	467
Строки документации: <code>__doc__</code>	469
PyDoc: функция <code>help</code>	472
PyDoc: отчеты в формате HTML	475
За рамками строк документации: Sphinx	483
Стандартный набор руководств	483
Веб-ресурсы	484
Изданные книги	485
Распространенные затруднения при написании кода	485
Резюме	487
Проверьте свои знания: контрольные вопросы	488
Проверьте свои знания: ответы	488
Проверьте свои знания: упражнения для части III	489
Часть IV. Функции и генераторы	491
ГЛАВА 16. Основы функций	492
Для чего используются функции?	493
Написание кода функций	494
Операторы <code>def</code>	496
Оператор <code>def</code> исполняется во время выполнения	496
Первый пример: определения и вызовы	497
Определение	497
Вызов	497
Полиморфизм в Python	498
Второй пример: пересечение последовательностей	499
Определение	500
Вызов	500
Еще раз о полиморфизме	501
Локальные переменные	502

Резюме	502
Проверьте свои знания: контрольные вопросы	503
Проверьте свои знания: ответы	503
ГЛАВА 17. Области видимости	504
Основы областей видимости в Python	504
Детали, касающиеся областей видимости	505
Распознавание имен: правило LEGB	507
Пример области видимости	510
Встроенная область видимости	511
Оператор <code>global</code>	514
Проектирование программы: минимизируйте количество глобальных переменных	515
Проектирование программы: минимизируйте количество межфайловых изменений	516
Другие способы доступа к глобальным переменным	518
Области видимости и вложенные функции	519
Детали вложенных областей видимости	519
Примеры вложенных областей видимости	519
Фабричные функции: замыкания	520
Сохранение состояния из объемлющей области видимости с помощью стандартных значений	523
Оператор <code>nonlocal</code> в Python 3.X	527
Основы оператора <code>nonlocal</code>	527
Оператор <code>nonlocal</code> в действии	529
Для чего используются оператор <code>nonlocal</code> ? Варианты сохранения состояния	531
Состояние с помощью оператора <code>nonlocal</code> : только Python 3.X	531
Состояние с помощью глобальных переменных: только одиночная копия	532
Состояние с помощью классов: явные атрибуты (предварительный обзор)	533
Состояние с помощью атрибутов функций: Python 3.X и 2.X	534
Резюме	538
Проверьте свои знания: контрольные вопросы	539
Проверьте свои знания: ответы	540
ГЛАВА 18. Аргументы	541
Основы передачи аргументов	541
Аргументы и разделяемые ссылки	542
Избегайте модификации изменяемых аргументов	544
Эмуляция выходных параметров и множественных результатов	545
Специальные режимы сопоставления аргументов	546
Основы сопоставления аргументов	547
Синтаксис сопоставления аргументов	548
Особенности использования специальных режимов сопоставления	549
Примеры ключевых слов и стандартных значений	550
Примеры произвольного количества аргументов	552
Аргументы с передачей только по ключевым словам Python 3.X	557
Функция <code>min</code>	560
Основная задача	561
Дополнительные очки	562

Заключение	563
Обобщенные функции для работы с множествами	563
Эмуляция функции print из Python 3.X	565
Использование аргументов с передачей только по ключевым словам	567
Резюме	569
Проверьте свои знания: контрольные вопросы	569
Проверьте свои знания: ответы	570
ГЛАВА 19. Расширенные возможности функций	571
Концепции проектирования функций	571
Рекурсивные функции	573
Суммирование с помощью рекурсии	574
Альтернативные варианты кода	574
Операторы цикла или рекурсия	576
Обработка произвольных структур	576
Объекты функций: атрибуты и аннотации	580
Косвенные вызовы функций: “первоклассные” объекты	580
Интроспекция функций	581
Атрибуты функций	582
Аннотации функций в Python 3.X	583
Анонимные функции: выражения lambda	585
Основы выражения lambda	586
Для чего используется выражение lambda?	587
Как (не) запутать свой код на Python	589
Области видимости: выражения lambda также могут быть вложенными	590
Инструменты функционального программирования	591
Отображение функций на итерируемые объекты: map	592
Выбор элементов из итерируемых объектов: filter	594
Комбинирование элементов из итерируемых объектов: reduce	594
Резюме	596
Проверьте свои знания: контрольные вопросы	596
Проверьте свои знания: ответы	596
ГЛАВА 20. Включения и генераторы	598
Списковые включения и инструменты функционального программирования	598
Списковые включения или map	599
Добавление проверок и вложенных циклов: filter	600
Пример: списковые включения и матрицы	603
Не злоупотребляйте списковыми включениями: KISS	605
Генераторные функции и выражения	608
Генераторные функции: yield или return	608
Генераторные выражения: итерируемые объекты встречаются с включениями	614
Генераторные функции или генераторные выражения	618
Генераторы являются объектами с одиночной итерацией	620
Генерация во встроенных типах, инструментах и классах	623
Пример: генерация перемешанных последовательностей	626
Не злоупотребляйте генераторами: EIBTI	631
Пример: эмуляция zip и map с помощью итерационных инструментов	633

Сводка по синтаксису включений	639
Области видимости и переменные включений	639
Осмысление включений множеств и словарей	641
Расширенный синтаксис включений для множеств и словарей	642
Резюме	642
Проверьте свои знания: контрольные вопросы	643
Проверьте свои знания: ответы	643
ГЛАВА 21. Оценочные испытания	645
Измерение времени выполнения итерационных альтернатив	645
Модуль измерения времени: любительский	646
Сценарий измерения времени	651
Результаты измерения времени	652
Альтернативные версии модуля для измерения времени	655
Другие варианты	658
Измерение времени выполнения итераций и версий Python с помощью модуля <code>timeit</code>	659
Базовое использование <code>timeit</code>	659
Модуль и сценарий оценочных испытаний: <code>timeit</code>	664
Результаты запуска сценария оценочных испытаний	666
Продолжаем забавляться с оценочными испытаниями	668
Другие темы, связанные с оценочными испытаниями: тест <code>pystone</code>	672
Затруднения, связанные с функциями	673
Локальные имена распознаются статически	673
Стандартные значения и изменяемые объекты	675
Функции без операторов <code>return</code>	677
Прочие затруднения, связанные с функциями	677
Резюме	678
Проверьте свои знания: контрольные вопросы	678
Проверьте свои знания: ответы	679
Проверьте свои знания: упражнения для части IV	679
Часть V. Модули и пакеты	683
ГЛАВА 22. Модули: общая картина	684
Для чего используются модули?	684
Архитектура программы Python	685
Структурирование программы	686
Импортирование и атрибуты	686
Стандартные библиотечные модули	688
Как работает импортирование	689
1. Поиск файла модуля	689
2. Компиляция файла модуля (возможная)	690
3. Выполнение файла модуля	691
Файлы байт-кода: <code>__pycache__</code> в Python 3.2+	691
Модели файлов байт-кода в действии	692
Путь поиска модулей	693
Конфигурирование пути поиска	696

Вариации пути поиска	696
Список <code>sys.path</code>	697
Выбор файла модуля	698
Резюме	700
Проверьте свои знания: контрольные вопросы	701
Проверьте свои знания: ответы	701
ГЛАВА 23. Основы написания модулей	702
Создание модулей	702
Имена файлов модулей	702
Другие виды модулей	703
Использование модулей	703
Оператор <code>import</code>	703
Оператор <code>from</code>	704
Оператор <code>from *</code>	704
Операции импортирования происходят только однократно	705
Операторы <code>import</code> и <code>from</code> являются присваиваниями	706
Эквивалентность <code>import</code> и <code>from</code>	707
Потенциальные затруднения, связанные с оператором <code>from</code>	708
Пространства имен модулей	709
Файлы генерируют пространства имен	709
Словари пространств имен: <code>__dict__</code>	711
Уточнение имен атрибутов	712
Импортирование или области видимости	712
Вложение пространств имен	713
Перезагрузка модулей	714
Основы использования <code>reload</code>	715
Пример использования <code>reload</code>	716
Резюме	718
Проверьте свои знания: контрольные вопросы	718
Проверьте свои знания: ответы	719
ГЛАВА 24. Пакеты модулей	720
Основы импортирования пакетов	721
Пакеты и настройки пути поиска	721
Файлы <code>__init__.py</code> пакетов	722
Пример импортирования пакетов	724
Использование <code>from</code> или <code>import</code> с пакетами	726
Для чего используется импортирование пакетов?	727
История о трех системах	727
Относительное импортирование пакетов	730
Изменения в Python 3.X	731
Основы относительного импортирования	732
Для чего используются операции относительного импортирования?	733
Границы действия операций относительного импортирования	736
Сводка по правилам поиска модулей	736
Операции относительного импортирования в действии	737
Затруднения, связанные с операциями импортирования относительно пакетов: смешанное использование	742

Пакеты пространств имен, введенные в Python 3.3	748
Семантика пакетов пространств имен	749
Влияние на обычные пакеты: необязательность <code>__init__.py</code>	750
Пакеты пространств имен в действии	751
Вложение пакетов пространств имен	752
Файлы по-прежнему имеют приоритет над каталогами	753
Резюме	756
Проверьте свои знания: контрольные вопросы	756
Проверьте свои знания: ответы	756
ГЛАВА 25. Расширенные возможности модулей	758
Концепции проектирования модулей	758
Сокрытие данных в модулях	760
Сведение к минимуму вреда от <code>from *: X и __all__</code>	760
Включение будущих языковых средств: <code>__future__</code>	761
Смешанные режимы использования: <code>__name__</code> и <code>__main__</code>	762
Модульное тестирование с помощью <code>__name__</code>	763
Пример: код с двойным режимом	764
Символы валют: Unicode в действии	767
Строки документации: документация по модулям в работе	769
Изменение пути поиска модулей	770
Расширение <code>as</code> для операторов <code>import</code> и <code>from</code>	771
Пример: модули являются объектами	772
Импортирование модулей по строкам с именами	775
Выполнение строк с кодом	775
Прямые вызовы: два варианта	776
Пример: транзитивная перезагрузка модулей	777
Инструмент рекурсивной перезагрузки	777
Альтернативные реализации	780
Затруднения, связанные с модулями	784
Конфликты имен модулей: операции импортирования пакетов и относительно пакетов	784
Порядок следования операторов в коде верхнего уровня имеет значение	785
Оператор <code>from</code> копирует имена, но не ссылки на них	786
Форма оператора <code>from *</code> может сделать неясным смысл переменных	787
Функция <code>reload</code> может не оказывать влияния на результаты операторов импортирования <code>from</code>	787
<code>reload</code> , <code>from</code> и тестирование в интерактивном сеансе	788
Рекурсивные операции импортирования <code>from</code> могут не работать	789
Резюме	790
Проверьте свои знания: контрольные вопросы	791
Проверьте свои знания: ответы	791
Проверьте свои знания: упражнения для части V	792
ПРИЛОЖЕНИЕ. Решения упражнений, приводимых в конце частей	794
Предметный указатель	819

Посвящается Вере. Ты моя жизнь.

Об авторе

Марк Лутц – ведущий инструктор по Python, автор самых ранних и ставших бестселлерами книг, посвященных Python, и новаторская личность в мире Python.

Марк является автором книг *Learning Python* (<http://learning-python.com/books/about-lp5e.html>), *Programming Python* (<http://learning-python.com/books/about-pp4e.html>) и *Python Pocket Reference* (<http://learning-python.com/books/about-pyref4e.html>) издательства O'Reilly, которые дожили до четвертых или пятых изданий. Он использовал и продвигал Python, начиная с 1992 года, приступил к написанию книг по Python в 1995 году (<http://learning-python.com/books>) и открыл курсы по обучению Python в 1997 году (<http://www.learning-python.com/index.html>). К середине 2019 года Марк провел 260 курсов обучения Python для более 4 000 студентов, а также написал 14 книг по Python, проданных в количестве свыше 675 000 экземпляров и переведенных, по меньшей мере, на десяток языков.

Усилия, приложенные Марком на ниве Python в течение двух десятилетий (<http://learning-python.com/books/about-python.html>), помогли упрочить его репутацию как одного из самых широко используемых языков программирования во всем мире. Вдобавок Марк работал в сфере программного обеспечения более 30 лет. Он получил степени бакалавра и магистра в области компьютерных наук в Висконсинском университете, где исследовал реализации языка Prolog, и на протяжении своей карьеры как профессионального разработчика программного обеспечения трудился над компиляторами, средствами программирования, сценарными приложениями и смешанными клиент-серверными системами.

Марк ведет обучающий веб-сайт (<http://learning-python.com/training>) и дополнительный веб-сайт для поддержки книг (<http://learning-python.com/books>).

Об иллюстрации на обложке

Животное, изображенное на обложке книги – древесная крыса (*Neotoma Muridae*). Древесная крыса проживает в самых разнообразных условиях (главным образом в скалистой и пустынной местности и в местности, покрытой низкорослой растительностью) на большей части Северной и Центральной Америки, как правило, поодаль от людей. Древесные крысы хорошо умеют лазить и гнездятся на деревьях или кустарниках вплоть до шестиметровой высоты от поверхности земли; некоторые виды роют норы под землей или прячутся в расщелинах скал либо занимают покинутые норы других видов.

Эти серовато-бежевые среднего размера грызуны являются незаурядными барахольщиками: они тащат в свои дома все, что находят, нужно оно им или нет, и особенно неравнодушны к блестящим предметам, таким как жестянные банки, стекло и столовое серебро.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой уничтожения; все они важны для нашего мира. Чтобы узнать больше о том, чем вы можете помочь, посетите веб-сайт animals.oreilly.com.

Изображение на обложке воспроизводит гравюру XIX века из иллюстраций в книге *Cuvier's Animal Kingdom*.

Предисловие

Если вы находитесь в книжном магазине и ищете краткую историю об этой книге, то вот и она.

- *Python* – мощный компьютерный язык программирования, поддерживающий множество парадигм, который оптимизирован для обеспечения высокой производительности программистов, читабельности кода и качества программного обеспечения.
- Эта книга предлагает всестороннее и доскональное введение в сам язык Python. Ее цель в том, чтобы помочь вам справиться с основами Python, прежде чем переходить к их применению в своей работе. Подобно всем предшествующим изданиям книга направлена на то, чтобы служить единым всеобъемлющим обучающим ресурсом для всех новичков в Python, будут они использовать Python 2.X, Python 3.X или обе линейки.
- *Настоящее издание* было написано во времена версий Python 3.3 и 2.7 (и в основном уточнено с учетом текущей для середины 2019 года версии Python 3.7 – *п rim.пер.*), а также в значительной степени расширено, чтобы отражать общепринятую практику, сложившуюся в мире Python.

В данном предисловии более подробно описаны цели, границы и структура этой книги. Читать его необязательно, но оно поможет сориентироваться до того, как вы приступите к чтению книги в целом.

“Экосистема” этой книги

Python представляет собой популярный язык программирования с открытым кодом, применяемый для написания автономных программ и сценарных приложений в широком разнообразии предметных областей. Он является бесплатным, переносимым, мощным, а также относительно легким и удивительно приятным в использовании. Программисты из всех уголков индустрии программного обеспечения считают, что ориентация Python на высокую продуктивность разработчиков и качество программного обеспечения должна быть стратегическим преимуществом как в крупных, так и в малых проектах.

Независимо от того, новичок вы в программировании или же профессиональный разработчик, эта книга направлена на то, чтобы ознакомить вас с языком Python быстрее, чем могут более ограниченные подходы. После чтения книги вы должны знать о языке Python достаточно для его применения в любых выбранных прикладных областях.

Книга задумывалась как руководство, в котором особый акцент делается на самом языке *Python*, а не на его специфических приложениях. Она входит в следующий набор:

- *Learning Python* (<http://www.oreilly.com/catalog/9781449355739>) – обучает самому языку Python с концентрацией внимания на основах языка, которые охватывают все предметные области;
- *Programming Python* (<http://www.oreilly.com/catalog/9780596158101>) показывает то, что можно делать с помощью Python после его изучения.

Такое разделение труда неслучайно. В то время как прикладные цели могут варьироваться от читателя к читателю, потребность в практическом обзоре основ языка остается неизменной. Книги, ориентированные на приложения, такие как *Programming Python*, логично продолжают данную книгу, используя реалистично масштабированные примеры для исследования роли Python в распространенных прикладных областях, среди которых веб-сеть, графические пользовательские интерфейсы, программные системы, базы данных и текст. Кроме того, книга *Python Pocket Reference* (<http://www.oreilly.com/catalog/9780596009403>) предлагает не включенные здесь справочные материалы и задумана как дополнение настоящей книги.

Однако поскольку книга сосредоточена на основах, появляется возможность представить принципы Python глубже, чем многие программисты могут увидеть при начальном изучении языка. Принятый в ней восходящий подход и самостоятельные учебные примеры рассчитаны на то, чтобы постепенно обучать читателей всему языку.

Знания базового языка, получаемые в процессе чтения, применимы к любой программной системе Python, которая вам встретится – будь это популярный инструмент вроде Django, NumPy и App Engine или другая система.

Будучи основанной на трехдневном учебном курсе языка Python с повсеместными контрольными вопросами и упражнениями, книга также служит введением в язык, позволяющим каждому самостоятельно выбирать скорость изучения. Хотя ее формат лишен живого взаимодействия, принятого на занятиях, это компенсируется добавочной глубиной и гибкостью, которую способна обеспечить только книга. Несмотря на то что книгу можно использовать многими способами, последовательные читатели сочтут ее приблизительно эквивалентной семестровому курсу лекций по Python.

О пятом издании

Предыдущее *четвертое издание* книги, вышедшее в 2009 году, охватывало версии Python 2.6 и 3.0¹. В нем рассматривались многие и временами несовместимые изменения, внесенные в линейку Python 3.X. Также было предложено новое руководство по объектно-ориентированному программированию и новые главы по более сложным темам, таким как текст Unicode, декораторы и метаклассы.

¹ Недолго просуществовавшее третье издание книги, вышедшее в 2007 году, охватывало версию Python 2.5 и ее более простой и узкий мир Python с единственной линейкой. История этой книги доступна по ссылке <http://learning-python.com/books>. С течением лет книга выросла в размере и сложности прямо пропорционально собственному росту Python. Согласно приложению В только в версии Python 3.0 появилось 27 дополнений и 57 изменений языка, которые были отражены в книге, а в Python 3.3 тенденция продолжилась. В наши дни программисты на Python сталкиваются с двумя несовместимыми линейками, тремя основными парадигмами, изобилием расширенных инструментов и порядочной избыточностью функциональным средств, большинство из которых не удастся аккуратно разделить между линейками Python 2.X и Python 3.X. Ситуация не настолько обескураживающая, как может показаться (многие инструменты являются просто вариациями на какую-то тему), но она полностью отражена в этой всеобъемлющей книге по Python.

Пятое издание, законченное в 2013 году, является пересмотром предыдущего издания, обновленным для охвата версий *Python 3.3* и *2.7*, текущих на то время выпусков в линейках *Python 3.X* и *Python 2.X* (оно в основном уточнено с учетом текущей для середины 2019 года версии *Python 3.7* – *прим.пер.*). Пятое издание включает все изменения языка в каждой линейке и доработано, чтобы улучшить представление материала. Ниже описаны основные моменты.

- Обзор *Python 2.X* обновлен для охвата таких средств, как включения словарей и множеств, которые раньше были доступны только в *Python 3.X*, но перенесены в *Python 2.7*.
- Обзор *Python 3.X* дополнен рассмотрением нового синтаксиса *yield* и *raise*; модели байт-кода *_runcache_*; пакетов пространств имен *Python 3.3*; режима с единственным браузером в *PyDoc*; изменений в литералах и хранении; а также нового запускающего модуля *Windows*, появившегося в версии *Python 3.3*.
- Добавлены новые или расширены прежние *смешанные* темы, такие как *JSON*, *timeit*, *PyPy*, *os.popen*, генераторы, рекурсии, слабые ссылки, *_mro_*, *_iter_*, *super*, *_slots_*, метаклассы, дескрипторы, *random*, *Sphinx* и т.д. Кроме того, в целом повышена совместимость с *Python 2.X* как в примерах, так и в изложении материала.

В этом издании появилось новое *заключение* в виде главы 41 (об эволюции *Python*), два новых *приложения* (о последних изменениях в *Python* и новом запускающем модуле *Windows*), а также новая *глава* (об оценочных испытаниях: расширенный пример измерения времени выполнения кода). Краткая сводка по *изменениям Python* приведена в приложении В второго тома. Там также подводятся итоги по отличиям между линейками *Python 2.X* и *Python 3.X* в целом, которые впервые рассматривались в предыдущем издании, хотя часть из них вроде классов нового стиля охватывают обе линейки и просто стали обязательными в *Python 3.X* (смысл *X* вскоре будет объяснен).

Согласно последнему пункту в предыдущем списке данное издание также получило определенный прирост, потому что оно полнее раскрывает более *сложные возможности языка*, которые на протяжении последнего десятилетия многие из нас всячески старались игнорировать как необязательные, но теперь они становятся более популярными в коде *Python*. Как мы увидим, эти инструменты делают *Python* более мощным, однако также поднимают планку для новичков и способны расширить границы и определение *Python*. Поскольку вы можете столкнуться с любым из них, я непосредственно раскрываю такие инструменты в книге, а не притворяюсь, что их не существует.

Несмотря на обновления, пятое издание сохраняет большую часть структуры и содержания предыдущего издания и по-прежнему предназначено служить исчерпывающим обучающим ресурсом по линейкам *Python 2.X* и *Python 3.X*. В то время как оно ориентируется главным образом на пользователей *Python 3.3* (3.7) и *Python 2.7*, исторический ракурс также делает его подходящим для *более старых* версий *Python*, которые регулярно применяются в наши дни.

Хотя будущее предсказывать невозможно, книга делает акцент на основах, которые были действительными на протяжении более двух десятилетий и вероятно будут применимы также к *будущим* версиям *Python*. Пробелы в книге, которые могут возникнуть с выходом новых версий, восполнят документы о нововведениях в комплекте руководств по *Python*.

Линейки Python 2.X и Python 3.X

Так как это сильно влияет на содержание книги, мне необходимо заранее сказать несколько слов об истории Python 2.X/3.X. Когда в 2009 году было написано *четвертое издание* книги, Python совсем недавно стал доступным в двух разновидностях:

- версия 3.0 была первой в линейке развивающейся и несовместимой мутации языка, в общем известной как *Python 3.X*;
- версия 2.6 сохранила обратную совместимость с огромной массой существующего кода Python и была последней в линейке, коллективно известной как *Python 2.X*.

Наряду с тем, что Python 3.X был в значительной степени тем же самым языком, он почти не запускал код, написанный для предшествующих выпусков. Он:

- навязал модель Unicode с обширными последствиями для строк, файлов и библиотек;
- превратил роль итераторов и генераторов во всепроникающую как часть более полной функциональной парадигмы;
- сделал обязательными классы нового стиля, которые объединяются с типами, но становятся более мощными и сложными;
- изменил многие фундаментальные инструменты и библиотеки, а также полностью заменил или удалил другие.

Единственная мутация `print` из оператора в функцию без преувеличения нарушила работу почти всех написанных ранее программ Python. Помимо стратегического потенциала обязательные модели Unicode и классов и вездесущие генераторы Python 3.X были ориентированы на отличающуюся практику программирования.

Хотя многие оценили Python 3.X как усовершенствование и как будущее Python, линейка Python 2.X все еще очень широко использовалась и долго поддерживалась параллельно с линейкой Python 3.X. Большинство кода Python было написано с применением Python 2.X и миграция на Python 3.X казалась медленным процессом.

Современная история Python 2.X/3.X

При написании *пятого издания* книги в 2013 году появились версии Python 3.3 (Python 3.7 на момент выхода русскоязычного издания в 2019 году) и Python 2.7, но история с Python 2.X/3.X по большому счету *не изменилась*. По существу теперь Python представляет собой мир с двумя версиями, в котором многие пользователи запускают обе линейки Python 2.X и Python 3.X согласно своим программным целям и зависимостям. При этом для многих новичков выбор между Python 2.X и Python 3.X остается выбором существующего программного обеспечения или передового языка. Несмотря на то что в Python 3.X было перенесено немало основных пакетов Python, многие другие пакеты в настоящее время по-прежнему работают только в Python 2.X.

Для ряда наблюдателей Python 3.X теперь выглядит как *несочинца*, позволяющая исследовать новые идеи, тогда как Python 2.X рассматривается как *испытанный Python*, который не обладает всеми возможностями Python 3.X, но является намного более распространенным. Другие все еще видят Python 3.X как будущее – представление, которое поддерживалось основными планами разработки в 2013 году: Python 2.7 продолжит поддерживаться, но будет последней версией в линейке Python 2.X, в то время как Python 3.3 станет очередной версией в развитии линейки Python 3.X.

С другой стороны, инициативы вроде *PyPy* (реализация Python 2.X, которая предлагает ошеломляющие улучшения производительности) представляют будущее Python 2.X, если только не откровенное отделение.

Несмотря на все мнения, по прошествии почти пяти (уже более десяти – *прим. нер.*) лет после своего выпуска Python 3.X пока еще не заменил собой Python 2.X. Как показатель, Python 2.X по-прежнему загружается чаще Python 3.X для Windows из сайта python.org вопреки тому факту, что данная оценка естественным образом искается в пользу новых пользователей и *самого недавнего* выпуска. Конечно, такие статистические данные подвержены изменениям, но прошедшие пять лет показательны в смысле внедрения Python 3.X. Существующая программная база Python 2.X все еще намного превосходит языковые расширения Python 3.X. Кроме того, последняя позиция в линейке Python 2.X делает версию Python 2.7 своего рода *стандартом де-факто*, невосприимчивым к постоянному темпу изменений в линейке Python 3.X – положительная характеристика для тех, кто ищет стабильную базу, и отрицательная для тех, кому важен непрерывный рост.

Лично я думаю, что современный мир Python достаточно велик, чтобы дать пристанище *обеим* линейкам Python 3.X и Python 2.X. Похоже, они удовлетворяют разным целям и привлекательны для разных сторон, что обеспечивает превосходство перед другими семействами языков (скажем, C и C++ давно существуют, хотя возможно отличаются между собой больше, чем Python 2.X и Python 3.X). Вдобавок, поскольку две линейки Python настолько подобны, навыки, обретенные в результате изучения любой из двух линеек Python, почти полностью переносятся на другую, особенно если вы прибегали к помощи ресурсов, охватывающих две версии, таких как эта книга. Фактически до тех пор, пока вы понимаете, в чем они расходятся, часто есть возможность писать код, который выполняется в обеих линейках.

В то же самое время такое разделение становится значительной дилеммой, не поддающей признаков ослабления, как для программистов, так и для авторов книг. Хотя при написании книги было бы легче притвориться, что Python 2.X не существует, и раскрывать только Python 3.X, это не будет отвечать потребностям крупной пользовательской базы Python, имеющейся на сегодняшний день. С применением Python 2.X был написан огромный объем кода, который в ближайшее время никуда не денется. И наряду с тем, что некоторые новички в языке могут и должны концентрироваться на Python 3.X, любой, кому приходится использовать код, написанный в прошлом, обязан быть на связи с нынешним миром Python 2.X. Так как могут пройти годы, пока многие сторонние библиотеки и расширения будут перенесены в Python 3.X, вполне возможно, что данная ветвь окажется не совсем временной.

Раскрытие линеек Python 3.X и Python 2.X

Чтобы учесть такое разветвление и отвечать потребностям всех потенциальных читателей, книга была обновлена для раскрытия обеих линеек Python 3.X и Python 2.X. Она ориентирована на программистов, применяющих Python 2.X, программистов, использующих Python 3.X, и программистов, привязанных к двум линейкам.

То есть книгу можно применять для изучения *любой из двух* линеек Python. Хотя часто делается особый акцент на Python 3.X, попутно также отмечаются отличия и инструменты Python 2.X, рассчитанные на программистов, которые используют более старый код. Наряду с тем, что две версии в основном похожи, в ряде важных аспектов они расходятся, на что и указывается в подходящих ситуациях.

Например, в большинстве примеров я буду применять вызовы `print` из Python 3.X, но также опишу оператор `print` из Python 2.X, так что вы сможете понять смысл

раннего кода и нередко использовать переносимые приемы вывода, которые выполняются в обеих линейках. Кроме того, я буду вводить новые средства, такие как оператор `nonlocal` в Python 3.X и строковый метод `format`, который доступен, начиная с версий Python 2.6 и Python 3.0, а также отмечать, когда подобные расширения отсутствуют в старых версиях Python.

Опосредованно это издание относится и к другим версиям Python 2.X/3.X, хотя в некоторых более старых версиях Python 2.X может не получиться запустить все примеры из книги. Скажем, несмотря на то, что декораторы классов доступны, начиная с версий Python 2.6 и Python 3.0, вы не можете применять их в более старых версиях Python 2.X, которые еще не располагали таким средством. Сводку по изменениям, внесенным в Python 2.X и Python 3.X, ищите в приложении В второго тома.

Какая версия Python должна использоваться?

Выбор версии может диктоваться вашей организацией, но если вы новичок в Python и учитесь самостоятельно, то можете задаться вопросом, какую версию устанавливать. Ответ зависит от ваших целей. Ниже приведено несколько советов по выбору.

Когда выбирать Python 3.X: новые возможности, развитие

Если вы впервые изучаете Python и не нуждаетесь в работе с любым существующим кодом Python 2.X, то я рекомендую начать с линейки Python 3.X. В ней устраниены давнишние недостатки и убран устаревший хлам, но одновременно сохранены все первоначальные основные идеи и добавлены элегантные новые инструменты. Например, бесшовная модель Unicode и более широкое применение генераторов и методик функционального программирования в Python 3.X многими пользователями рассматривается как ценное свойство. Многие популярные библиотеки и инструменты Python уже доступны для Python 3.X или будут доступны ко времени чтения вами книги, особенно принимая во внимание постоянное усовершенствование линейки Python 3.X. Все развитие языка происходит только в Python 3.X, что приводит к добавлению новых средств и совершенствованию Python, но также превращает определение языка в постоянно движущуюся цель — компромисс, присущий переднему краю.

Когда выбирать Python 2.X: существующий код, стабильность

Если вы будете использовать систему, основанную на Python 2.X, то линейка Python 3.X в текущий момент может не подойти. Тем не менее, вы обнаружите, что книга также решит ваши проблемы и поможет перейти на Python 3.X в будущем. Вдобавок вы окажетесь в большой компании. Все группы, которые я обучал в 2012 году, применяли только Python 2.X, и по-прежнему встречается полезное программное обеспечение на Python в форме, предназначеннной только для Python 2.X. Кроме того, в отличие от Python 3.X линейка Python 2.X больше не будет изменяться — что в зависимости от обстоятельств расценивается как достоинство или как недостаток. В использовании и написании кода Python 2.X нет ничего плохого, но у вас может быть желание следить за линейкой Python 3.X и ее продолжающимся развитием. Будущее Python еще предстоит написать, и во многом оно зависит от пользователей, включая вас.

Когда выбирать обе линейки: код, нейтральный к версиям

Вероятно, лучшая новость здесь заключается в том, что главные принципы Python остаются одинаковыми в обеих линейках – Python 2.X и Python 3.X отличаются в аспектах, которые многие пользователи считут незначительными, и книга призвана помочь освоить обе линейки. На самом деле при условии понимания отличий между ними зачастую легко писать код, нейтральный к версиям, который выполняется под управлением обеих линеек Python, что будет постоянно встречаться в этой книге. Ищите в приложении В второго тома указания по миграции Python 2.X/3.X и советы относительно написания кода для обеих линеек и аудитории.

Независимо от того, на какой версии или версиях вы сосредоточите внимание поначалу,обретенные вами навыки будут переноситься прямо туда, куда приведет работа с Python.



Замечание относительно применения X. Для ссылки на совокупные выпуски, представленные в двух линейках, будут использоваться формы Python 3.X и Python 2.X. Скажем, *Python 3.X* включает версии от 3.0 до 3.3/3.7 и будущие выпуски, а *Python 2.X* означает все версии, начиная с 2.0 и заканчивая 2.7 (и, по всей видимости, никаких других). Более конкретные выпуски упоминаются, когда предмет обсуждения применим только к ним (например, литералы множеств в Python 2.7 или запускающий модуль и пакеты пространств имен в Python 3.3). Иногда это замечание окажется слишком общим, поскольку некоторые средства, помеченные здесь как относящиеся к Python 2.X, могут отсутствовать в ранних выпусках Python 2.X, редко используемых в наши дни, но оно подогнано к линейке Python 2.X, которой насчитывается без малого 20 лет.

Предпосылки и усилия

Установить абсолютные предпосылки для этой книги невозможно, т.к. ее полезность и ценность могут зависеть от мотивации читателя в той же степени, что и от имеющихся у него знаний. И настоящие новички, и подлинные ветераны в программировании в прошлом успешно использовали данную книгу. Если вы заинтересованы в изучении Python и хотите уделить ему соответствующее время и внимание, тогда книга наверняка вам пригодится.

Сколько времени потребуется, чтобы изучить Python? Несмотря на то что затраты будут варьироваться от ученика к ученику, книга оказывает наибольшее воздействие, когда ее читают. Некоторые читатели могут применять книгу как справочный ресурс, но большинство людей, стремящихся к совершенному владению Python, должны ожидать затрат по меньшей мере недель и возможно месяцев на проработку материалов книги в зависимости от того, насколько тщательно они отслеживают предлагаемые примеры. Как уже упоминалось, книга является приблизительным эквивалентом семестрового курса по самому языку Python.

Такова оценка времени изучения только самого Python и обретения навыков, требуемых для его эффективного использования. Хотя этой книги может оказаться достаточно для достижения основных целей написания сценариев, читатели, которые надеются заняться разработкой программного обеспечения во всем объеме, должны ожидать, что после прочтения данной книги им придется выделить дополнительное время на освоение более крупномасштабных проектов и возможно обратиться к книгам вроде *Programming Python* (<http://www.oreilly.com/catalog/9780596158101>).

Новость может не выглядеть приятной для тех, кто надеется на мгновенное получение квалификации, но программирование – непростое занятие (вопреки тому, что вы могли слышать о нем). Современный язык Python и программное обеспечение в целом являются достаточно сложными, чтобы заслуживать усилий, потраченных на освоение таких всеобъемлющих книг, как эта. Ниже приведено несколько указаний по работе с книгой для читателей, обладающих и не обладающих опытом программирования.

Опытные программисты

Вы имеете начальное преимущество и можете быстрее пройти ряд начальных глав, но вы не должны упустить из виду основные идеи и вам вполне вероятно придется избавиться от некоторого багажа. В общем случае исследование любых приемов программирования или написания сценариев до настоящей книги может быть полезным по причине вероятных аналогий. С другой стороны, я также обнаружил, что предшествующий опыт программирования может стать помехой из-за предположений, укоренившихся в других языках (бывших программистов на Java или C++ очень легко распознать по коду Python, который они пишут поначалу!). Правильное применение Python требует принятия надлежащего типа мышления. За счет концентрации на основных концепциях данная книга призвана помочь вам научиться писать код на Python в духе Python.

Настоящие новички

Вы тоже в состоянии изучать здесь язык Python, равно как и само программирование, но возможно придется работать несколько усерднее и дополнять книгу другими вводными ресурсами. Даже если вы еще не считаете себя программистом, то книга все равно будет для вас полезной, но вероятно темп ее изучения окажется медленнее, к тому же в ходе чтения понадобится тщательно прорабатывать все примеры и упражнения. Также имейте в виду, что в книге уделяется больше внимания обучению самому языку Python, а не основам программирования. Если вы обнаруживаете, что не понимаете материал, тогда я рекомендую предварительно ознакомиться с какой-нибудь вводной книгой по программированию и только затем приниматься за эту книгу. На веб-сайте Python есть много ссылок на полезные ресурсы для начинающих.

Формально настоящая книга призвана служить *первой книгой по Python для любого рода новичков*. Она может не быть идеальным ресурсом для того, кто никогда раньше не прикасался к компьютеру (например, в ней не тратится время на выяснение, что собой представляет компьютер), но и не делается особенно много предположений относительно вашего опыта или образования в области программирования.

С другой стороны, я не буду обижать читателей, считая их “пустышками”, что бы под этим ни понималось – посредством Python легко делать полезные вещи и в книге будет показано, каким образом. В книге Python иногда противопоставляется с такими языками, как C, C++, Java и прочими, но вы можете благополучно игнорировать такие сравнения, если в прошлом не пользовались другими языками.

Структура этой книги

Чтобы помочь вам сориентироваться, в этом разделе предлагается краткое изложение содержания и целей главных частей книги. Если вам не терпится добраться до них, тогда можете спокойно пропустить данный раздел (или взамен просмотреть ог-

лавление). Однако краткая дорожная карта для настолько объемной книги некоторым читателям наверняка покажется достоинством.

По замыслу каждая *часть* раскрывает значительную функциональную область языка и состоит из *глав*, сконцентрированных на специфических темах или аспектах этой области. Вдобавок каждая глава заканчивается *контрольными вопросами* и ответами на них, а каждая часть – более крупными *упражнениями*, решения которых представлены в приложении.



Практика имеет значение. Я настоятельно рекомендую читателям по возможности прорабатывать контрольные вопросы и упражнения, предложенные в книге, и разбирать примеры в целом. В программировании ничто не способно заменить опробование прочитанного на практике. Делаете вы это с данной книгой или с собственным проектом, но фактическое написание кода имеет решающее значение, если вы хотите придерживаться изложенных здесь идей.

В целом презентация книги будет восходящей, т.к. таков характер Python. По мере продвижения примеры и темы становятся все более сложными. Скажем, классы Python по большому счету являются всего лишь пакетами функций, которые обрабатывают встроенные типы. Как только вы освоите встроенные типы и функции, классы потребуют относительно небольшого мысленного скачка. Поскольку каждая часть построена на основе предшествующих, большинство читателей считут, что *последовательное чтение* имеет наибольший смысл. Ниже приведен обзор главных частей книги, с которыми вам предстоит ознакомиться. 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, раскрытие которых 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.

Чем эта книга не является

Учитывая накопленную с годами довольно крупную читательскую аудиторию, неизбежно появлялись и те, кто ожидал, что книга исполнит роль, выходящую за пределы ее границ. Итак, после того, как было указано, чем эта книга является, необходимо прояснить, чем она *не является*.

- Книга представляет собой обучающее руководство, а *не* справочник.
- В книге раскрывается сам язык, а *не* приложения, стандартные библиотеки или сторонние инструменты.
- Книга предлагает всесторонний взгляд на значительную тему, а *не* расплывчатый обзор.

Поскольку перечисленные пункты являются ключевыми для содержания книги, имеет смысл раскрыть их более подробно.

Это не справочник и не руководство по специфическим приложениям

Книга является *обучающим руководством по языку*, а не справочником и не книгой о приложениях. Так задумано: *современный язык Python* с его встроенными типами, генераторами, замыканиями, включениями, Unicode, декораторами и сочетанием парадигм процедурного, объектно-ориентированного и функционального программирования превращает один лишь основной язык в значительную тему, знание которой становится необходимым условием вашей будущей работы с Python независимо от того, какой предметной областью вы будете заниматься. Тем не менее, когда вы готовы к ознакомлению с другими ресурсами, то вот несколько советов и напоминаний.

Справочные ресурсы

Как следует из предыдущего описания структуры книги, для поиска информации вы можете использовать предметный указатель и содержание, но никаких справочных приложений в книге не предусмотрено. Если вас интересуют справочные ресурсы по Python (что очень скоро случится у большинства читателей), то я рекомендую книгу *Python Pocket Reference*, <http://www.oreilly.com/catalog/9780596009403/> (*Python. Кафманский справочник, 5-е изд*, <http://www.williamspublishing.com/Books/978-5-8459-1912-0.html>), а также другие справочники, которые легко найти, и стандартные справоч-

ные руководства по Python, поддерживаемые на <http://www.python.org>. Последние бесплатны, всегда актуальны и доступны как в онлайновом режиме, так и в виде устанавливаемой версии для Windows.

Приложения и библиотеки

Как обсуждалось ранее, книга также не является руководством по специфическим *приложениям* вроде программирования для веб-сети, построения графических пользовательских интерфейсов и разработки систем. Опосредованно это включает библиотеки и инструменты, применяемые при работе над приложениями; хотя в ней представлены некоторые *стандартные библиотеки* и инструменты, в том числе `timeit`, `shelve`, `pickle`, `struct`, `json`, `pdb`, `os`, `urllib`, `re`, `xml`, `random`, *PyDoc* и *IDLE*, они не входят в основные границы ее охвата. Если вы ищете более полное раскрытие таких тем и уже хорошо знаете Python, тогда я рекомендую среди прочих почитать книгу *Programming Python* (<http://www.oreilly.com/catalog/9780596158101>), но сначала вы обязаны получить устойчивое представление об основном языке. В инженерной области, подобной разработке программного обеспечения, нужно сначала научиться ходить, а затем уже бегать.

Это не краткая история для спешащих людей

По объему книги легко понять, что она не экономит на деталях: книга представляет *полный язык Python*, а не краткий обзор упрощенного подмножества языковых средств. Попутно в ней также раскрываются *программные принципы*, которые важны для написания хорошего кода Python. Как упоминалось ранее, это книга для изучения на протяжении многих недель или месяцев, предназначенная для передачи уровня квалификации, который был бы получен в результате прослушивания полноценного курса лекций по Python.

Так тоже сделано намеренно. Конечно, многим читателям книги не требуются навыки полномасштабной разработки программного обеспечения, а некоторые могут осваивать Python постепенно. Вместе с тем, поскольку в коде, который вы будете встречать, может использоваться *любая* часть языка, ни одна часть не будет в полном смысле необязательной для большинства программистов. Кроме того, даже те, кто пишет сценарии от случая к случаю, и те, кто занимается Python в качестве хобби, должны знать базовые принципы разработки программного обеспечения, чтобы кодировать правильно и надлежащим образом применять готовые инструменты.

Эта книга призвана удовлетворять обе указанные потребности, т.е. язык и принципы, достаточно глубоко, чтобы быть полезной. Однако в конечном итоге обнаружится, что более развитые инструменты Python, такие как поддержка объектно-ориентированного и функционального программирования, относительно легко освоить, справившись с их предварительными условиями — и вы справитесь, если будете прорабатывать книгу по одной главе за раз.

Изложение последовательно до той степени, до которой позволяет Python

Говоря о *порядке чтения*, в этом издании также предпринята попытка свести к минимуму *ссылки вперед*, но изменения, внесенные в Python 3.X, в некоторых случаях делают их неизбежными (на самом деле иногда Python 3.X похоже предполагает, что вы уже знаете Python, хотя вы только его изучаете!).

Вот лишь несколько типичных примеров.

- Вывод, сортировки, строковый метод `format` и ряд вызовов `dict` полагаются на *ключевые аргументы*.
- Списки и проверки словарных ключей, а также вызовы `list`, используемые многими инструментами, заключают в себе концепции *имперации*.
- Применение `exec` для выполнения кода теперь предполагает наличие знаний *файловых объектов и интерфейсов*.
- Написание кода для обработки новых *исключений* требует знания *классов* и основ объектно-ориентированного программирования.
- И так далее – даже элементарное наследование поднимает более сложные темы, такие как *метаклассы и дескрипторы*.

Язык Python по-прежнему лучше всего изучать как прогрессию от простого к сложному, и последовательное чтение здесь все еще наиболее благоразумно. Тем не менее, некоторые темы могут требовать перепрыгивания и произвольного поиска. Чтобы свести это к минимуму, в книге будут указываться ссылки вперед, когда они случаются, но их влияние максимально смягчено.



Но если у вас мало времени. Хотя при изучении Python важна глубина, некоторые читатели могут располагать лишь ограниченным временем. Если вы заинтересованы в том, чтобы начать с быстрого тура по Python, то я рекомендую прочитать главы 1, 4, 10 и 28 (и возможно 26) – краткий обзор, который наверняка вызовет интерес к более полной истории, изложенной в остальных главах книги и необходимой большинству читателей. Книга намеренно *разделена на уловни*, чтобы облегчить освоение материала – с введениями, за которыми следуют детали, так что вы можете начать с обзоров и со временем погружаться глубже. Вам не обязательно читать эту книгу всю сразу, но принятый в ней постепенный подход призван помочь в конечном итоге разобраться с ее материалами.

Программы в книге

В целом я стремился в книге придерживаться независимого подхода в отношении версий Python и платформ. Книга задумана так, чтобы быть полезной пользователям всех версий Python. И все же из-за того, что Python с течением времени меняется, а платформы отличаются с практической точки зрения, имеет смысл описать специфические системы, которые вы увидите в действии в большинстве примеров.

Версии Python

Все примеры программ основаны на версиях Python 3.3/3.7 и 2.7. Кроме того, многие примеры запускаются под управлением выпусков, предшествующих Python 3.X/2.X, и в ходе изложения даются примечания об истории изменения языка для пользователей более старых версий Python.

Однако поскольку книга сконцентрирована на ядре языка, вы можете быть уверены в том, что большинство приведенных здесь деталей не особенно сильно изменится в будущих выпусках Python, как отмечалось ранее. Большая часть книги применима также к *ранним версиям Python* кроме случаев, когда это не так; естественно, если вы попытаетесь использовать расширения, добавленные после выхода версии, с которой работаете, то успеха не добьетесь. В качестве эмпирического правила запомните: при наличии возможности модернизации наилучшей версией Python будет самая последняя.

Так как внимание в книге сосредоточено на ядре языка, большинство материала применимо к *Jython* и *IronPython* (реализациям языка Python, основанным на Java и .NET), а также к другим реализациям Python, таким как *Stackless* и *PyPy* (рассматриваются в главе 2). Упомянутые альтернативы отличаются в основном деталями использования, но не языком.

Платформы

Рассмотренные в книге примеры запускались на компьютере с *Windows 7* и *8*, хотя переносимость Python снижает важность данного момента особенно с учетом того, что книга ориентирована на основы. Вы заметите несколько особенностей, связанных с Windows, включая окна командной строки, экранные снимки, указания по установке и появившийся в версии Python 3.3 запускающий модуль Windows, но это отражает лишь тот факт, что большинство новичков, скорее всего, будут начинать с платформы Windows, и может быть благополучно проигнорировано пользователями других операционных систем.

Я также предоставляю несколько деталей запуска для других платформ вроде Linux, такие как применение строки `#!`, но в главе 3 и в приложении Б второго тома вы увидите, что запускающий модуль Windows, введенный в версии Python 3.3, даже это делает более переносимым приемом.

Загрузка кода примеров для книги

Исходный код примеров, рассмотренных в книге, а также решений предложенных упражнений, доступен для загрузки в форме файла ZIP по ссылке <http://oreil.ly/LearningPython-5E> и на веб-сайте издательства.

Разумеется, примеры лучше прорабатывать одновременно с чтением книги, к тому же вам понадобятся базовые знания по запуску программ на Python. Подробности начального запуска приведены в главе 3.

Использование кода, сопровождающего книгу

Код в моих книгах по Python предназначен для обучающих целей, и я буду рад, если он поможет читателям в таком качестве. Само издательство O'Reilly придерживается официальной политики относительно повторного использования кода примеров, рассмотренных в книге, которая сформулирована ниже.

Настоящая книга призвана помочь вам выполнять свою работу. Обычно если в книге предлагается пример кода, то вы можете применять его в собственных программах и документации. Вы не обязаны обращаться к нам за разрешением, если только не используете значительную долю кода. Скажем, написание программы, в которой задействовано несколько фрагментов кода из этой книги, разрешения не требует. Для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение обязательно. Ответ на вопрос путем цитирования данной книги и ссылки на пример кода разрешения не требует. Для встраивания значительного объема примеров кода, рассмотренных в этой книге, в документацию по вашему продукту разрешение обязательно.

Мы высоко ценим указание авторства, хотя и не требуем этого. Установление авторства обычно включает название книги, фамилию и имя автора, издательство и номер ISBN. Например: "Learning Python, Fifth Edition, by Mark Lutz. Copyright 2013 Mark Lutz, 978-1-4493-5573-9".

Если вам кажется, что способ использования вами примеров кода выходит за законные рамки или упомянутые выше разрешения, тогда свяжитесь с нами по следующему адресу электронной почты: permissions@oreilly.com.

Соглашения, используемые в этой книге

В книге приняты следующие типографские соглашения.

Курсив

Применяется для новых терминов.

Моноширинный

Используется для программного кода, содержимого файлов и вывода команд, а также для обозначения модулей, методов, операторов и системных команд.

Моноширинный полужирный

Применяется в разделах кода для обозначения текста или команд, которые должны быть набраны пользователем, и временами для выделения порций кода.

Моноширинный курсив

Используется для заменяемых фрагментов и ряда комментариев в коде.



Здесь приводится совет, указание или общее замечание, связанное с близлежащим текстом.



Здесь приводится предупреждение или предостережение, относящееся к близлежащему тексту.

Вы также будете периодически сталкиваться с *врезками* и *сносками*, которые часто необязательны для чтения, но дополнительно проясняют ситуацию по обсуждаемой теме. Во врезках, посвященных применению, таких как врезка “Что потребует внимания: срезы” в главе 7, нередко приводятся примеры сценариев использования для исследуемых средств.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@dialektika.com

WWW: <http://www.dialektika.com>

Благодарности

Так как в 2013 году я пишу уже пятое издание этой книги, мне трудно удержаться от взгляда в прошлое. Я использовал и продвигал Python в течение 21 года, писал книги о нем 18 лет и обучал группы вживую на протяжении 16 лет. Несмотря на пройденное время, я по-прежнему постоянно поражаюсь успехам, которых достиг Python – в областях, которые большинство из нас не могли себе даже представить в начале 1990-х годов. Поэтому, рискуя походить на безнадежно эгоцентричного автора, я надеюсь, что вы простите мне несколько заключительных слов касательно истории и благодарностей.

Предыстория

Моя собственная история, связанная с Python, предшествовала Python 1.0 и веб-сети (и восходит к тому времени, когда установка подразумевала извлечение вложений из сообщений электронной почты, их объединение, раскодирование и упование на то, что все это каким-то образом заработает). Когда я впервые открыл для себя Python как несостоявшийся разработчик программного обеспечения на C++ в 1992 году, то понятия не имел о том, каким образом он повлияет на следующие два десятилетия моей жизни. Через два года после выхода первого издания книги *Programming Python*, посвященной Python 1.3, в 1995 году я начал путешествовать по стране и миру, обучая Python новичков и экспертов. Завершив написание первого издания книги *Learning Python* в 1999 году, я стал независимым инструктором и писателем книг по Python, отчасти благодаря феноменальному росту его популярности.

И вот результат. К настоящему времени я написал 13 книг по Python (5 изданий этой книги и по 4 издания двух других), которых по моим данным было продано около 400 000 экземпляров (по состоянию на середину 2019 года 14 книг и 675 000 проданных экземпляров – *прим. пер.*). Я обучал языку Python более 15 лет, провел 260 обучающих курсов в США, Европе, Канаде и Мексике, а также выпустил примерно 4 000 студентов. Помимо движения к несбыточной мечте стать пассажиром, часто летающим самолетами, обучающие курсы помогли мне улучшить как эту, так и другие мои книги по Python. Обучение оттачивало книги и наоборот, результатом чего стало то, что мои книги близко отражают происходящее в группах студентов и могут служить для них жизнеспособной альтернативой.

Что касается самого языка Python, то в последние годы он вошел в число от 5 до 10 наиболее широко применяемых языков программирования в мире (по разным источникам и в разное время). Состояние Python будет исследоваться в первой главе книги, поэтому остаток истории я продолжу там.

Благодарности Python

Поскольку процесс обучения учит инструкторов обучать, эта книга многим обязана моим *обучающим курсам*. Я хотел бы поблагодарить всех *студентов*, посещавших мои курсы в течение последних 16 лет. Наряду с изменениями в самом Python ваши отзывы сыграли важную роль в приведении в порядок материала книги; нет ничего более поучительного, чем воочию наблюдать за тем, как 4 000 человек повторяют одни и те же ошибки начинающих! Изменения, внесенные в последние издания данной книги, обязаны своим появлением в первую очередь недавним курсам, хотя каждый курс, проводимый с 1997 года, в определенной степени помог улучшить книгу. Я хотел бы поблагодарить тех, кто выделял помещения для проведения курсов в Дублине,

Мехико, Барселоне, Лондоне, Эдмонтоне и Пуэрто-Рико; такой опыт был одним из самых значительных достижений в моей карьере.

Из-за того, что процесс написания учит инструкторов писать, книга также многим обязана своей *читательской аудитории*. Я хочу выразить благодарность бесчисленным читателям, которые в течение последних 18 лет находили время, чтобы помочь советами, как в онлайновом режиме, так и лично. Ваши отзывы были жизненно важны для развития этой книги и стали значимым фактором ее успеха — преимущества, которые присущи миру открытого кода. Комментарии читателей включали в себя весь спектр от “Вам следовало бы вообще запретить писать книги” до “Премного благодарен вам за написание этой книги”; если в таких вещах и возможен консенсус, то вероятно он находится где-то посередине, хотя я перефразирую Толкина: книга все еще слишком коротка.

Я также хотел бы выразить признательность всем, кто принимал участие в *производстве* данной книги. Всем, кто помогал сделать книгу надежным продуктом на многие годы — научным и художественным редакторам, маркетологам, техническим рецензентам и многим другим. Самому издательству O'Reilly за то, что мне был предоставлен шанс принять участие в 13 проектах по написанию книг; это было весело (и лишь немного напоминало фильм “День сурка”).

Дополнительная благодарность всему *обществу Python*; подобно большинству систем с открытым кодом Python является продуктом многих невоспетых усилий. Для меня было честью наблюдать, как Python вырос из младенца в семье сценарных языков до широко используемого инструмента, который в определенном виде развернут почти в каждой организации, занимающейся разработкой программного обеспечения. Оставив в стороне формальные расхождения, это был захватывающий процесс, заслуживающий того, чтобы стать его частью.

Я также хочу поблагодарить своего первоначального редактора в O'Reilly, покойного Фрэнка Уиллисона. Эта книга в значительной степени была идеей Фрэнка. Он оказал глубокое влияние и на мою карьеру, и на успех языка Python, когда он был в новинку — наследие, о котором я вспоминаю каждый раз, когда возникает искушение неправильно употребить слово “только”.

Личные благодарности

И напоследок несколько личных благодарностей. Покойному Карлу Сагану за то, что вдохновил 18-летнего парня из Висконсина. Мое матери за мужество. Моим близким родственникам за поиск истины. Книге *The Shallows* за крайне необходимое предупреждение.

Моему сыну Майклу и дочерям Саманте и Роксане за то, что вы такие, какие есть. Я не уверен, что заметил, когда вы выросли, но горжусь тем, как вы это сделали, и стремлюсь увидеть, куда жизнь поведет вас дальше.

Моей жене Вере за терпение, стойкость, диетическую колу и претцели. Я рад, что наконец нашел тебя. Я не знаю, что нас ждет в ближайшие 50 лет, но я надеюсь провести их все вместе с тобой.

Марк Лутц, весна 2013 года

ЧАСТЬ I

Начало работы

Python в вопросах и ответах

Если вы приобрели эту книгу, то вероятно знаете, что собой представляет Python и почему он является важным инструментом, подлежащим изучению. В противном случае вам вряд ли удастся зарабатывать, программируя на языке Python, пока вы не изучите его, прочитав остаток книги и выполнив пару проектов. Но прежде чем погружаться в детали, в первой главе будут кратко представлены главные причины популярности Python. Чтобы положить начало определения Python, материал главы подается в форме наиболее распространенных вопросов со стороны новичков вместе с ответами на них.

Почему люди используют Python?

Из-за доступности в наши дни настолько большого количества языков это обычный первый вопрос, задаваемый новичками. Принимая во внимание, что в настоящие времена насчитывается приблизительно миллион пользователей Python, ответить на данный вопрос совершенно точно не получится; иногда выбор инструментов разработки основан на уникальных ограничениях или личных предпочтениях.

Но после обучения на протяжении прошедших 16 лет примерно 260 групп, насчитывающих в сумме свыше 4 000 студентов, у меня сформировались некоторые основные мысли. Ниже перечислены главные факторы, на которые ссылаются пользователи Python.

Качество программного обеспечения

Для многих концентрация Python на читабельности, согласованности и качестве программного обеспечения (ПО) в целом отличает его от других инструментов в мире языков написания сценариев. Код Python по замыслу должен быть *читабельным*, а потому многократно используемым и сопровождаемым — в гораздо большей степени, чем традиционные языки написания сценариев. Согласованность кода Python облегчает его понимание, даже если он написан не вами. Вдобавок Python располагает развитой поддержкой механизмов многократного применения ПО, таких как объектно-ориентированное и функциональное программирование.

Производительность труда разработчиков

Python повышает производительность труда разработчиков во много раз по сравнению с компилируемыми и статически типизированными языками вроде C, C++ и Java. Код Python обычно занимает *от одной трети до одной пятой* части размера эквивалентного кода C++ или Java. В итоге приходится меньше набирать на клавиатуре, меньше отлаживать и меньше впоследствии сопровождать. Кроме того, программы Python запускаются немедленно, без длительных шагов компиляции и связывания, требующихся в ряде других инструментов, что дополнительно увеличивает скорость работы программистов.

Переносимость программ

Большинство программ Python функционирует без изменений на *всех основных компьютерных платформах*. Например, перенос кода Python между Linux и Windows, как правило, сводится к копированию кода сценария между машинами. Более того, Python предлагает многочисленные варианты для написания кода графических пользовательских интерфейсов, программ доступа к базам данных, веб-систем и т.д. Даже интерфейсы операционных систем, включая запуск программ и обработку каталогов, являются в Python насколько возможно переносимыми.

Поддерживающие библиотеки

Вместе с Python поставляется большая коллекция предварительно собранной и переносимой функциональности, которая называется *стандартной библиотекой*. Стандартная библиотека поддерживает множество решений программных задач прикладного уровня, начиная с сопоставления текста с образцом и заканчивая сценариями для сетей. В добавок Python можно расширять библиотеками собственной разработки и обширным набором прикладного поддерживающего ПО, созданного сторонними разработчиками. *Область стороннего ПО* для Python предлагает инструменты, предназначенные для конструирования веб-сайтов, численного программирования, доступа к последовательным портам, разработки игр и многое другое (позже будут приведены примеры). Скажем, расширение NumPy было представлено как бесплатный и более мощный эквивалент системы численного программирования Matlab.

Интеграция компонентов

В сценариях Python можно легко взаимодействовать с другими частями приложения, используя различные механизмы интеграции. Такая интеграция позволяет применять Python в качестве инструмента для *настройки и расширения* продуктов. В настоящее время из кода Python можно обращаться к библиотекам C и C++, его можно вызывать из программ C и C++, интегрировать с компонентами Java и .NET, взаимодействовать с фреймворками вроде COM и Silverlight, сопрягаться с устройствами через последовательные порты и взаимодействовать через сети с помощью интерфейсов, подобных SOAP, XML-RPC и CORBA. Он не является автономным инструментом.

Наслаждение

Легкость использования Python и наличие встроенного инструментального набора позволяет сделать процесс программирования в *большой степени приятным, нежели рутинным*. И хотя наслаждение от программирования можно отнести к нематериальным преимуществам, его влияние на производительность работы оказывается важным качеством.

Из всех упомянутых факторов первые два (качество и продуктивность) являются, пожалуй, наиболее убедительными для большинства пользователей Python и заслуживают исчерпывающего описания.

Качество программного обеспечения

По замыслу Python реализует преднамеренно простой и читабельный синтаксис, а также чрезвычайно согласованную модель программирования. Как подтверждает слоган на недавней конференции, посвященной Python, общий результат заключается в том, что Python, кажется, “вписывается в ваш мозг” – т.е. средства языка взаимодействуют согласованными и ограниченными путями и естественным образом вытекают из небольшого набора основных концепций. В итоге язык становится более легким для изучения, понимания и запоминания. На практике при чтении или написании кода программистам Python нет необходимости постоянно обращаться к справочному руководству; это согласованно спроектированная система, которая, как многие находят, выдает удивительно единообразный код.

Что касается философии, то Python принимает до некоторой степени минималистский подход. Это означает, что хотя обычно доступно несколько способов решения задачи, связанной с написанием кода, как правило, имеется только один очевидный способ, немного менее очевидных альтернатив и небольшой набор согласованных взаимодействий повсюду в языке. Кроме того, Python не принимает произвольные решения вместо вас; когда взаимодействия неоднозначны, то явное вмешательство предпочтительнее “магии”. В образе мышления Python явное лучше неявного и простое лучше сложного¹.

Помимо принципов проектирования Python включает такие инструменты, как модули и объектно-ориентированное программирование (ООП), которое естественным образом содействует многократному использованию кода. И поскольку Python концентрируется на качестве, то на нем естественным образом сосредоточены также программисты на Python.

Продуктивность труда разработчиков

Во время резкого подъема развития Интернета, длившегося с середины до конца 1990-х годов, было трудно отыскать достаточное число программистов для реализации программных проектов; разработчикам предлагали реализовывать системы с той же скоростью, с какой развивался Интернет. В более позднюю эру сокращений и экономического спада картина изменилась. Программистов часто просили выполнять те же самые задачи с еще меньшим количеством людей.

В обоих сценариях Python зарекомендовал себя как блестящий инструмент, который позволял программистам делать больше с меньшими усилиями. Он преднамеренно оптимизирован для ускорения разработки – его простой синтаксис, динамическая типизация, отсутствие шагов компиляции и встроенный инструментальный набор

¹ Чтобы получить более полное представление о философии Python, введите в любом окне командной подсказки Python команду `import this` (в главе 3 будет показано как). В результате вызывается “информация-сюрприз”, скрытая в Python – коллекция принципов проектирования, положенных в основу Python, которые пронизывают язык и сообщество его пользователей. Среди них можно обнаружить сокращение EIBTI, обозначающее правило “explicit is better than implicit” (“явное лучше неявного”), которое теперь стало популярным жаргоном. Принципы культом не являются, но довольно близки к тому, чтобы считаться девизом и мировоззрением Python, на которые мы будем часто ссылаться в книге.

предоставляют программистам возможность разрабатывать программы за гораздо меньшее время, чем то, которое пришлось бы потратить, применяя ряд других инструментов. В итоге Python обычно поднимает продуктивность труда разработчиков до уровня, во много раз превышающего уровень продуктивности, поддерживаемые традиционными языками. Это хорошие новости как во время подъема, так и во время спада и для любого периода, когда индустрия ПО находится между ними.

Является ли Python “языком написания сценариев”?

Python является универсальным языком программирования, который часто выступает в ролях, связанных с написанием сценариев. Он обычно определяется как *объектно-ориентированный язык написания сценариев* – определение, которое комбинирует поддержку ООП с общей ориентацией на сценарные роли. Если выразиться одной строкой, то я бы сказал, что Python вероятно лучше известен как *универсальный язык программирования, чем смесь процедурной, функциональной и объектно-ориентированной парадигм* – формулировка, которая захватывает богатство и сферы применения современного языка Python.

Тем не менее, понятие “написание сценариев”, похоже, прилипло к Python как банный лист, возможно в качестве противопоставления с более крупными усилиями по программированию, требующимися другими инструментами. Например, для описания файла кода Python люди часто используют слово “сценарий” вместо “программа”. Соблюдая эту традицию, термины “сценарий” и “программа” в книге встречаются взаимозаменяемо с небольшим предпочтением в пользу термина “сценарий” при описании простого файла верхнего уровня и термина “программа” при ссылке на более сложное многофайловое приложение.

Однако из-за того, что понятие “язык написания сценариев” имеет так много значений для разных экспертов, некоторые из них предпочли бы вообще не применять его к Python. Фактически услышав, что Python называют языком написания сценариев, у людей обычно возникают три очень разных ассоциации, как вполне уместные, так и не особенно.

Инструменты командной оболочки

Когда Python описывается как язык написания сценариев, временами люди полагают, что он представляет собой инструмент для создания кода сценариев, ориентированных на операционную систему. Такие программы часто запускаются в командной строке консоли и выполняют задачи вроде обработки текстовых файлов и запуска других программ.

Программы Python могут и исполняют такие роли, но это лишь одна из десятков обычных предметных областей Python. Он не просто лучший язык написания сценариев для командной оболочки.

Язык управления

Для других написание сценариев относится к “связующему” уровню, используемому для контроля и управления другими компонентами приложения. Программы Python действительно часто развертываются в контексте более крупных приложений. Например, для тестирования аппаратных устройств про-

граммы Python могут обращаться к компонентам, которые обеспечивают низкоуровневый доступ к тому или иному устройству. Аналогично программы могут запускать фрагменты кода Python в контрольных точках, чтобы поддерживать настройку продукта для конечного пользователя без необходимости в поставке и перекомпиляции исходного кода всей системы.

Простота Python делает его гибким инструментом управления по своей природе. Тем не менее, формально это тоже всего лишь распространенная роль Python; многие (возможно и большинство) программисты Python пишут автономные сценарии, даже не применяя или не зная о каких-либо встроенных компонентах. Python – не просто язык управления.

Легкость использования

Вероятно, лучше всего считать, что понятие “язык написания сценариев” относится к простому языку, используемому для быстрого кодирования задач. Это особенно справедливо, когда понятие применяется к Python, который делает возможной гораздо более быструю разработку программ, чем компилируемые языки, подобные C++. Его ускоренный цикл разработки стимулирует исследовательский и пошаговый режим программирования, с которым необходимо хорошо освоиться, чтобы оценить.

Однако не обманывайтесь – язык Python предназначен не только для решения простых задач. Наоборот, благодаря присущей легкости использования и гибкости он делает задачи простыми. Python располагает простым набором функциональных возможностей, но при необходимости позволяет масштабировать программы в плане сложности. Из-за этого Python обычно применяется для решения быстрых тактических задач и долгосрочной стратегической разработки.

Итак, Python – язык написания сценариев или нет? Ответ зависит от того, у кого вы спрашиваете. В целом понятие “написание сценариев” наверно лучше всего использовать для описания быстрого и гибкого режима разработки, поддерживаемого Python, а не конкретной предметной области.

Хорошо, но в чем недостаток?

После 21-летнего применения языка, 18-летнего написания о нем и 16-летнего обучения ему я обнаружил, что единственный значительный недостаток текущей реализации Python связан с тем, что *скорость выполнения* может не всегда быть такой же, как у полностью компилируемых и низкоуровневых языков, подобных C и C++. Хотя и относительно редко в наши дни, но при решении некоторых задач может все же возникнуть необходимость “приблизиться к железу” за счет использования низкоуровневых языков вроде тех, что более прямо отображаются на лежащую в основе аппаратную архитектуру.

Мы будем рассматривать концепции реализации более подробно позже в книге. Выражаясь кратко, современные стандартные реализации Python компилируют (т.е. транслируют) операторы исходного кода в промежуточный формат, известный как *байт-код*, и затем интерпретируют этот байт-код. Байт-код обеспечивает переносимость, т.к. он представляет собой независимый от платформы формат. Тем не менее, поскольку код Python обычно не компилируется до машинного кода (например, до инструкций для процессора Intel), некоторые программы будут выполняться в Python медленнее, чем в полностью компилируемом языке наподобие С. Система PyPy, обсуж-

даемая в следующей главе, способна достичь ускорения в десять-сто раз при выполнении определенного кода за счет дополнительной компиляции в ходе запуска программы, но она является отдельной альтернативной реализацией.

Будет ли вас беспокоить разница в скорости выполнения, зависит от того, какие виды программ вы пишете. Python много раз подвергался оптимизации и в большинстве предметных областей код Python выполняется достаточно быстро сам по себе. К тому же всякий раз, когда вы делаете в сценарии Python что-то “реальное” вроде обработки файла или конструирования графического пользовательского интерфейса, программа в действительности будет выполняться со скоростью С, поскольку такие задачи немедленно привязываются к скомпилированному коду С внутри интерпретатора Python. По существу выигрыш в скорости разработки, обеспечиваемый Python, зачастую важнее, чем любые потери в скорости выполнения, особенно с учетом производительности современных компьютеров.

Даже принимая во внимание производительность современных процессоров, по-прежнему существуют предметные области, которые требуют оптимальных скоростей выполнения. Например, численное программирование и анимация часто нуждаются в том, чтобы, по крайней мере, их основные компоненты перемалывания чисел выполнялись со скоростью С (или быстрее).

Если вы работаете в такой предметной области, то все равно можете применять Python – просто вынесите части приложения, которые требуют оптимальной скорости, в *скомпилированные расширения*, и свяжите их со своей системой для использования в сценариях Python.

Расширения в настоящей книге рассматриваются не особенно широко, но представляют собой просто пример роли Python в качестве языка управления, обсуждавшейся ранее. Главным примером этой двуязычной стратегии является расширение численного программирования для Python под названием *NumPy*; за счет объединения скомпилированных и оптимизированных библиотек численных расширений с языком Python расширение NumPy превращает Python в инструмент численного программирования, который одновременно эффективен и прост в применении. При необходимости такие расширения предоставляют мощный инструмент оптимизации.

Другие компромиссы, связанные с Python: нематериальные аспекты

Я уже упоминал, что единственным значительным недостатком Python является скорость выполнения. Для многих пользователей Python, главным образом для новичков, это действительно так. Большинство людей находят Python легким в изучении и увлекательным в использовании, особенно в сравнении с его ровесниками наподобие Java, C# и C++. Однако в интересах полного раскрытия темы я обязан также отметить ряд более абстрактных компромиссов, которые мне приходилось наблюдать за два десятилетия пребывания в мире Python – как преподавателя и как разработчика.

Как преподаватель я иногда обнаруживал, что частота внесения изменений в язык Python и его библиотеки была негативной чертой, а временами сетовал на его *рост* с течением лет. Отчасти ситуация объяснялась тем, что преподаватели и авторы книг находятся на переднем крае таких вещей; моей работой было обучение языку, несмотря на его постоянное изменение – задача сродни описи разбегающихся котят! Тем не менее, это широко распространенная проблема. Как будет показано в книге, исходный лейтмотив Python “быть проще” в наши дни часто определяется тенденцией к более сложным решениям за счет крутой кривой обучения новичков. Косвенным доказательством такой тенденции является размер настоящей книги.

С другой стороны, согласно большинству оценок Python все еще гораздо проще своих альтернатив и возможно сложен лишь настолько, насколько он должен быть, чтобы соответствовать многочисленным ролям, которые он исполняет сегодня. Его общая согласованность и открытая природа остаются захватывающими характеристиками для большинства. Кроме того, не всем нужно быть в курсе последних достижений, что четко показывает текущая популярность версии Python 2.X.

Как разработчик я также порой подвергаю сомнению компромиссы, присущие *подходу "батарейки в комплекте"*, который принят в Python по отношению к разработке. Его акцент на предварительно собранных инструментах может добавить зависимости (что, если применяемая батарейка изменяется, повреждается или устаревает?) и подталкивать к созданию решений для частных случаев вместо использования общих принципов, которые в конечном итоге способны обслуживать пользователей лучше (как вы сможете оценить либо задействовать инструмент, если не понимаете его целевое назначение?). Вы найдете в этой книге примеры обеих проблем.

Для типичных пользователей, особенно для людей, увлеченных своим хобби, и начинающих, основным полезным качеством является подход с инструментальным набором Python. Но не удивляйтесь, когда вы перерастете готовые инструменты и сможете извлечь выгоду из тех видов навыков, которые стремится привить настоящая книга. Или, перефразируя пословицу: дайте людям инструмент, и они будут писать код весь день; научите их строить инструменты, и они будут писать код всю жизнь. Цель книги соответствует больше второму варианту, нежели первому.

Как упоминалось ранее в главе, и Python, и его модель инструментального набора также испытывают недостатки, общие для проектов с *открытым кодом* в целом — потенциальное торжество личного *предпочтения* меньшинства перед широким употреблением большинства и появление время от времени *анафии* и даже *элитарности*, хотя они считаются наиболее серьезными последствиями в современных новых выпусках.

Мы еще вернемся к ряду компромиссов ближе к концу книги, когда вы изучите Python достаточно хорошо для того, чтобы делать собственные выводы. Определение того, чем "является" Python как система с *открытым кодом*, возложено на его пользователей. В конце концов, в наши дни язык Python популярен как никогда ранее, и его рост не демонстрирует никаких признаков ослабления. Для некоторых это может быть более выразительным показателем, чем индивидуальные мнения, как за, так и против.

Кто использует Python в наши дни?

На момент написания главы наилучшая оценка размера пользовательской базы Python, которую, по всей видимости, может сделать любой, предполагала наличие приблизительно 1 миллиона пользователей по всему миру (плюс или минус сколько-то). Оценка основана на разнообразных статистических данных, таких как количество загрузок, веб-статистика и опросы разработчиков. Поскольку Python — продукт с *открытым кодом*, получить более точное число затруднительно, т.к. отсутствуют какие-либо регистрации лицензий, которые можно было бы подсчитать. Вдобавок Python автоматически входит в состав дистрибутивов Linux, поставляется с компьютерами Macintosh и включается в широкий диапазон продуктов и оборудования, что еще больше усложняет получение достоверного представления о пользовательской базе.

Однако в общем Python обладает крупной пользовательской базой и очень активным сообществом разработчиков. В целом считается, что на сегодняшний день Python входит в *пятерку* или *десятку* наиболее широко применяемых языков программирова-

ния в мире (его точное место варьируется в зависимости от источника и даты). По причине существования Python на протяжении свыше двух десятилетий и обширного использования он также характеризуется высокой стабильностью и надежностью.

Помимо применения индивидуальными пользователями Python также задействован в реальных продуктах, приносящих доход для реальных компаний. Например, вот что общеизвестно из пользовательской базы Python.

- *Google* всесторонне использует Python в своих системах веб-поиска.
- Популярная служба совместного использования видеоматериалов *YouTube* почти полностью написана на Python.
- Серверное ПО и ПО для настольных клиентов службы хранилища *Dropbox* написано главным образом на Python.
- Одноплатный компьютер *Raspberry Pi* поддерживает Python в качестве своего учебного языка.
- *EVE Online*, грандиозная многопользовательская онлайновая игра от CCP Games, широко применяет Python.
- Популярная пиринговая система обмена файлами *BitTorrent* начинала свое существование как программа Python.
- *Industrial Light & Magic*, *Pixar* и другие компании используют Python в производстве анимационных фильмов.
- *ESRI* применяет Python в качестве инструмента настройки для конечных пользователей в своих картографических продуктах ГИС.
- Фреймворк веб-разработки *App Engine* от Google использует Python как прикладной язык.
- Продукт почтового сервера *IronPort* для выполнения своей работы применяет более 1 строк кода Python.
- *Maya*, мощная интегрированная система трехмерного моделирования и анимации, предоставляет API-интерфейс для сценариев Python.
- *NSA* использует Python для шифрования и анализа разведывательной информации.
- *iRobot* применяет Python при разработке коммерческих и военных роботизированных устройств.
- Настраиваемые сценарные события в игре *Civilization IV* написаны целиком на Python.
- В проекте “один лэптоп на ребенка” (One Laptop Per Child – *OLPC*) пользовательский интерфейс и модель деятельности построены на Python.
- *Netflix* и *Yelp* документально подтвердили роль Python в своих инфраструктурах ПО.
- *Intel*, *Cisco*, *Hewlett-Packard*, *Seagate*, *Qualcomm* и *IBM* используют Python для аппаратного тестирования.
- *JPMorgan Chase*, *UBS*, *Getco* и *Citadel* применяют Python для выработки прогнозов на финансовом рынке.
- *NASA*, *Los Alamos*, *Fermilab*, *JPL* и другие компании используют Python для решения задач научного программирования.

И так далее – несмотря на то, что список весьма показателен, полная отчетность выходит за рамки настоящей книги, и почти гарантированно со временем будет меняться. Чтобы получить свежую информацию о дополнительных пользователях, приложениях и ПО на Python, посетите следующие страницы, которые в текущий момент находятся на веб-сайте Python и в англоязычной Википедии, а также воспользуйтесь поиском в своем веб-браузере:

- истории успеха – <http://www.python.org/about/success>;
- предметные области – <http://www.python.org/about/apps>;
- цитаты пользователей – <http://www.python.org/about/quotes>;
- страница в англоязычной Википедии – http://en.wikipedia.org/wiki/List_of_Python_software.

Вероятно единственная общая идея, которая объединяет все упомянутые выше компании, заключается в том, что язык Python задействован в самых разных предметных областях. Универсальная природа Python делает его применимым почти ко всем областям, а не к какой-то одной. На самом деле можно с уверенностью сказать, что по существу любая реальная организация, занимающаяся написанием ПО, использует Python, то ли для решения краткосрочных тактических задач, таких как тестирование и администрирование, то ли для долгосрочной стратегической разработки продуктов. Python доказал эффективную работу в обоих режимах.

Что можно делать с помощью Python?

В дополнение к тому, что Python является хорошо спроектированным языком программирования, он удобен при решении реальных задач – то, чем разработчики занимаются изо дня в день. Он обычно применяется в разнообразных областях в качестве инструмента для снабжения сценариями других компонентов и реализации автономных программ. Фактически роли Python как универсального языка практически безграничны: вы можете использовать его для чего угодно, начиная с разработки веб-сайтов и игровых программ и заканчивая управлением роботами и космическими аппаратами.

Тем не менее, в настоящее время самые распространенные роли языка Python относятся к нескольким обширным категориям. В последующих разделах описаны наиболее распространенные приложения Python, а также инструменты, применяемые в каждой области. Мы не в состоянии сколько-нибудь глубоко исследовать упоминаемые здесь инструменты – если вас интересует любая из этих тем, тогда обращайтесь за дополнительными деталями на веб-сайт Python или другие ресурсы.

Системное программирование

Встроенные интерфейсы Python для служб операционных систем делают его идеальным средством написания переносимых и сопровождаемых инструментов и утилит администрирования систем (иногда называемых *инструментами командной оболочки*). Программы Python могут производить поиск в файлах и деревьях каталогов, запускать другие программы, организовывать параллельную обработку с помощью процессов и потоков и т.д.

Стандартная библиотека Python поставляется с привязками POSIX и поддерживает все обычные инструменты ОС: переменные среды, файлы, сокеты, конвейеры, процессы, многопоточность, сопоставление с образцом в виде регулярного выражения,

аргументы командной строки, интерфейсы стандартных потоков данных, модули запуска команд оболочки, развертывание имен файлов, утилиты сжатия файлов, средства разбора XML и JSON, обработчики файлов CSV и т.д. Вдобавок основная масса системных интерфейсов Python спроектирована так, чтобы быть переносимыми; например, запуск сценария, который копирует деревья каталогов, обычно не требует изменений на всех главных платформах Python. Реализация *Stackless Python*, описанная в главе 2 и используемая *EVE Online*, также предлагает улучшенные решения для удовлетворения требований многопроцессорной обработки.

Графические пользовательские интерфейсы

Простота и быстрота разработки в Python также делают его подходящим вариантом для программирования настольных графических пользовательских интерфейсов. В состав Python входит стандартный объектно-ориентированный интерфейс к Tk GUI API под названием *tkinter* (*Tkinter* в 2.X), который позволяет программам Python реализовывать переносимые графические пользовательские интерфейсы с присущим данной системе внешним видом и поведением. Графические пользовательские интерфейсы Python/*tkinter* выполняются в неизменном виде в средах Microsoft Windows, X Window (в Unix и Linux) и Mac OS (Classic и OS X). Бесплатный пакет расширения *Pmw* добавляет в инструментальный набор *tkinter* более развитые виджеты (графические элементы). Кроме того, API-интерфейс для построения графических пользовательских интерфейсов *wxPython*, основанный на библиотеке C++, предлагает альтернативный инструментальный набор для конструирования переносимых графических пользовательских интерфейсов в Python.

Высокоуровневые инструментальные наборы вроде *Dabo* построены поверх базовых API-интерфейсов, подобных *wxPython* и *tkinter*. С подходящей библиотекой вы можете также применять поддержку графических пользовательских интерфейсов из других инструментальных наборов в Python, например, *Qt* с *PyQt*, *GTK* с *PyGTK*, *MFC* с *PyWin32*, *.NET* с *IronPython* и *Swing* с *Jython* (Java-версия Python, описанная в главе 2) или *JPyre*. Для приложений, выполняющихся в веб-браузерах либо имеющих простые требования к пользовательскому интерфейсу, веб-фреймворки *Jython* и *Python* и серверные CGI-сценарии, рассматриваемые в следующем разделе, предоставляют дополнительные варианты создания пользовательских интерфейсов.

Написание сценариев для Интернета

В состав Python входят стандартные модули для Интернета, которые позволяют программам Python выполнять широкий спектр задач, связанных с сетями, в клиентском и серверном режимах. Сценарии могут взаимодействовать через сокеты; извлекать информацию из форм, отправленных серверными CGI-сценариями; передавать файлы посредством FTP; разбирать и генерировать документы XML и JSON; посылать, получать, составлять и анализировать сообщения электронной почты; извлекать веб-страницы по URL-адресам; разбирать HTML-содержимое извлеченных веб-страниц; взаимодействовать через протоколы XML-RPC, SOAP и Telnet, а также многое другое. Библиотеки Python делают решение таких задач удивительно простым.

Кроме того, в веб-сети доступен крупный набор сторонних инструментов для Интернет-программирования на Python. Например, система *HTMLGen* генерирует HTML-файлы из описаний на основе классов Python, пакет *mod_python* эффективно запускает Python на веб-сервере Apache и поддерживает серверный механизм шаблонов с помощью Python Server Pages, а система *Jython* обеспечивает бесшовную интеграцию

Python/Java и поддерживает написание серверных аплетов, которые запускаются на стороне клиента.

В добавок пакеты развитых фреймворков веб-разработки для Python, такие как *Django*, *TurboGears*, *web2py*, *Pylons*, *Zope* и *WebWare*, поддерживают быстрое конструирование полнофункциональных и пригодных для производственного применения веб-сайтов на Python. Многие из них включают средства объектно-реляционного отображения, архитектуру “модель-представление-контроллер” (Model-View-Controller), написание серверных сценариев и шаблонов и поддержку AJAX, чтобы предлагать полные решения веб-разработки производственного уровня.

Недавно Python был расширен с целью охвата насыщенных Интернет-приложений (rich Internet application – RIA) посредством таких инструментов, как *Silverlight* в *IronPython* и *pyjs* (известный еще и под названием *pyjamas*) и его компилятора Python в JavaScript, фреймворк AJAX и набор виджетов. С помощью *App Engine* и других механизмов, рассматриваемых в разделе “Программирование для баз данных” далее в главе, Python был перемещен в облачные вычисления. Там, где лидирует веб-сеть, Python быстро догоняет.

Интеграция компонентов

Мы обсуждали интеграцию компонентов ранее, когда Python был описан как язык управления. Способность Python расширяться и встраиваться в системы C и C++ делает его удобным в качестве гибкого связующего языка для сценарного описания поведения других систем и компонентов. Скажем, интеграция библиотеки C в Python позволяет Python проверять наличие и запускать компоненты библиотеки, а встраивание Python в какой-то продукт делает возможным кодирование настроек на месте эксплуатации без потребности в перекомпиляции целого продукта (или вообще поставки его исходного кода).

Инструменты вроде генераторов кода *SWIG* и *SIP* могут автоматизировать порядочную часть работы по связыванию скомпилированных компонентов с Python для использования в сценариях, а система *Cython* позволяет программистам смешивать код Python и языков семейства C. Более масштабные фреймворки, подобные поддержке COM в Python для Windows, основанной на Java реализации Jython и основанной на .NET реализации IronPython, предоставляют альтернативные способы написания сценариев для компонентов. Например, в среде Windows сценарии Python могут задействовать фреймворки для взаимодействия с Word и Excel, доступа к *Silverlight* и т.п.

Программирование для баз данных

Для традиционных требований относительно баз данных в Python предусмотрены интерфейсы ко всем распространенным системам реляционных баз данных – Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite и т.д. В мире Python также определен *переносимый API-интерфейс к базам данных* для доступа к системам баз данных SQL из сценариев Python, который выглядит одинаково в различных системах баз данных. Скажем, из-за того, что интерфейсы поставщика реализуют переносимый API-интерфейс, сценарий, написанный для работы с бесплатной системой MySQL, будет практически без изменений работать с другими системами (вроде Oracle); обычно понадобится лишь заменить лежащий в основе интерфейс поставщика. Встроенный внутрипроцессный механизм баз данных SQL, *SQLite*, является стандартной частью Python, начиная с версии 2.5, и поддерживает как прототипирование, так и основные потребности программ в хранении данных.

Что касается области, не связанной с SQL, то стандартный модуль Python под названием `pickle` предлагает простую систему постоянства объектов, которая позволяет программам легко сохранять и восстанавливать объекты Python из файлов и подобных им объектов. В веб-сети вам удастся отыскать сторонние системы с открытым кодом *ZODB* и *Durus*, которые предоставляют законченные системы объектно-ориентированных баз данных для сценариев Python. Также вы найдете системы *SQLObject* и *SQLAlchemy*, реализующие средства объектно-реляционного отображения (*object relational mapper* – ORM), которые сопоставляют модель классов Python с реляционными таблицами. Наконец, вы обнаружите *PyMongo* – интерфейс к *MongoDB*, высокопроизводительной, с открытым кодом, отличной от SQL документной базе данных в стиле *JSON*. Данные в MongoDB хранятся в структурах, очень похожих на собственные списки и словари Python, текст которых может анализироваться и создаваться с помощью стандартного библиотечного модуля Python по имени `json`.

Есть также системы, предлагающие более специализированные способы хранения данных, в том числе *App Engine* от Google, которая моделирует данные с помощью классов Python и обеспечивает всестороннюю масштабируемость, а также дополнительно появляющиеся варианты облачного хранения, в частности *Azure*, *PiCloud*, *OpenStack* и *Stackato*.

Быстрое прототипирование

Для программ Python компоненты, написанные на Python и C, выглядят одинаковыми. В итоге становится возможным прототипирование систем изначально на Python с последующим переносом избранных компонентов в компилируемые языки типа C или C++ для поставки. В отличие от ряда инструментов прототипирования Python не требует полного переписывания после того, как прототип был сформирован. Ради облегчения сопровождения и использования те части системы, которым не требуется эффективность языков вроде C++, могут оставаться реализованными на Python.

Численное и научное программирование

Python также широко применяется в численном программировании – области, которая традиционно не считалась сферой действия языков написания сценариев, но стала одним из самых захватывающих вариантов использования Python. Хорошо заметное здесь высокопроизводительное расширение численного программирования для Python, *NumPy*, включает такие расширенные инструменты, как объекты массивов, интерфейсы к стандартным математическим библиотекам и многое другое. За счет интеграции Python с численными процедурами, ради высокой скорости написанными на компилируемом языке, расширение NumPy превращает Python в сложно устроенный, но легкий в применении инструмент численного программирования. Он часто способен заменить существующий код, который написан на традиционных компилируемых языках, таких как FORTRAN или C++.

Дополнительные инструменты численного программирования для Python поддерживают анимацию, трехмерную визуализацию, параллельную обработку и т.д. Например, популярные расширения *SciPy* и *ScientificPython* предлагают дополнительные библиотеки инструментов для научного программирования и в качестве основного компонента используют NumPy. Реализация PyPy языка Python (обсуждается в главе 2) также получила поддержку в области численного программирования, отчасти потому, что громоздкий алгоритмический код, характерный для этой области, может выполняться в PyPy значительно быстрее, зачастую в 10–100 раз.

И еще: игры, изображения, глубинный анализ данных, роботы, электронные таблицы Excel...

Python обычно применяется в большем перечне областей, чем можно раскрыть здесь. Например, вы обнаружите инструменты, которые позволяют использовать Python в следующих ситуациях:

- программирование игр и создание мультимедиа-содержимого с помощью *pygame*, *cgi*, *pyglet*, *PySoy*, *Panda3D* и т.п.;
- взаимодействие через последовательные порты в Windows, Linux и других средах с помощью расширения *PySerial*;
- обработка изображений посредством *PIL* и его новейшего ответвления *Pillow*, *PyOpenGL*, *Blender*, *Maya* и т.д.;
- программирование контроллеров для роботов с применением инструментального набора *PyRo*;
- обработка естественного языка с помощью пакета *NLT*K;
- инструментальное оснащение плат *Raspberry Pi* и *Arduino*;
- мобильные вычисления посредством переносимых версий Python для платформ Google *Android* и Apple *iOS*;
- программирование функций и макросов для электронных таблиц Excel с использованием дополнений *PyXLL* или *DataNira*;
- обработка меток содержимого медиафайлов и метаданных с помощью *PyMedia*, *ID3*, *PIL/Pillow* и т.д.;
- программирование искусственного интеллекта с применением библиотеки нейронных сетей *PyBrain* и инструментального набора машинного обучения *Milk*;
- программирование экспертных систем посредством *PyCLIPS*, *Pyke*, *Prolog* и *pyDatalog*;
- мониторинг сети с использованием системы *zenoss*, написанной и настраиваемой с помощью Python;
- проектирование и моделирование, подкрепленное сценариями Python, с использованием *PythonCAD*, *PythonOCC*, *FreeCAD* и т.д.;
- обработка и генерация документов посредством *ReportLab*, *Sphinx*, *Cheetah*, *PyPDF* и т.д.;
- визуализация данных с применением *Mayavi*, *matplotlib*, *VTK*, *VPython* и т.д.;
- разбор XML-содержимого с помощью библиотечного пакета *xml*, модуля *xmlrpclib* и сторонних расширений;
- обработка файлов JSON и CSV с использованием модулей *json* и *csv*;
- глубинный анализ данных посредством фреймворка *Orange*, пакета *Pattern*, *Scrapy* и специального кода.

С помощью программы *PySolFC* вы даже можете раскладывать пасьянс. И, разумеется, вы можете писать специальные сценарии в областях, не настолько переполненных модными словечками, чтобы выполнять повседневное администрирование системы, обрабатывать электронную почту, управлять библиотеками документов и медиадан-

ных и т.п. Вы найдете ссылки на поддержку во многих областях на веб-сайте PyPI и через поиск (ищите ссылки в Google или на <http://www.python.org>).

Несмотря на широкое практическое применение, многие из этих специфических областей по большому счету представляют собой просто очередные случаи проявления роли интеграции компонентов, исполняемой Python. Добавление его в качестве внешнего интерфейса к библиотекам компонентов, написанных на компилируемом языке вроде C, делает Python удобным для написания сценариев в самых разных предметных областях. Как универсальный язык, который поддерживает интеграцию, Python обладает широкой применимостью.

Как Python разрабатывается и поддерживается?

Будучи популярной системой с открытым кодом, Python имеет крупное и активное сообщество разработчиков, реагирующее на проблемы и вносящее улучшения со скоростью, которую многие разработчики коммерческого ПО сочли бы поразительной. Разработчики Python координируют работу в онлайновом режиме с помощью системы управления версиями. Изменения разрабатываются согласно формальному протоколу, который включает написание *предложения по улучшению Python* (Python Enhancement Proposal – PEP) или другого документа и расширение системы регрессионного тестирования Python. Фактически модификация Python в наши дни примерно так же сложна, как изменение коммерческого ПО, что сильно отличается от разработки Python на ранних этапах, когда достаточно было написать электронное сообщение его создателю, но это хорошо, принимая во внимание величину пользовательской базы Python в настоящее время.

Фонд программного обеспечения Python (Python Software Foundation – PSF), официальная некоммерческая группа, организует конференции и занимается вопросами интеллектуальной собственности. Многочисленные конференции, посвященные Python, проводятся по всему миру; крупнейшими считаются OSCON, которую организует O'Reilly, и PyCon, проводимая PSF. Первая касается множества проектов с открытым кодом, а вторая представляет собой событие, связанное только с Python, популярность которого в последние годы значительно возросла. Конференции PyCon 2012 и PyCon 2013 насчитывали до 2 500 участников каждая; по сути, на PyCon 2013 следовало было ограничить лимит на этом уровне после неожиданного аншлага на PyCon 2012 (где удалось привлечь широкое внимание как к техническим, так и к нетехническим темам, которые я здесь не затрагиваю). В предшествующие годы посещаемость часто удваивалась, скажем, от 586 участников в 2007 году до свыше 1 000 участников в 2008 году, что в целом свидетельствовало о росте популярности Python и производило глубокое впечатление на тех, кто помнил ранние конференции, участники которых по большому счету могли уместиться за одним столиком в ресторане.

Компромиссы, связанные с открытым кодом

Нельзя не отметить тот факт, что хотя Python обладает энергичным сообществом разработчиков, это связано с неотъемлемыми компромиссами. ПО с открытым кодом может также выглядеть хаотичным, временами даже напоминать *анафхию* и не всегда быть гладко реализованным, как могло вытекать из разделов выше. Некоторым изменениям все же удается проигнорировать официальные протоколы, и как во всех человеческих начинаниях, несмотря на контроль процесса, по-прежнему могут возникать ошибки (например, версия Python 3.2.0 поступила с неработающей консольной функцией `input` для Windows).

Кроме того, в проектах с открытым кодом коммерческая выгода обменивается на личные предпочтения текущего круга разработчиков, которые могут совпадать или не совпадать с вашими предпочтениями — вы не являетесь заложником компании, но во власти тех, у кого есть свободное время для изменения системы. Конечный результат заключается в том, что развитие ПО с открытым кодом часто направляется немногими, но навязывается многим.

Однако на практике эти компромиссы гораздо больше влияют на тех, кто чрезсурп быстро переходит на новые выпуски, нежели на тех, кто пользуется устоявшимися версиями системы, включая предыдущие выпуски Python 3.X и Python 2.X. Например, если вы продолжали работать с традиционными классами из Python 2.X, то были практически невосприимчивы к бурному росту функциональности и изменений в классах нового стиля, которые происходили с начала до середины 2000-х годов. Хотя они стали обязательными в Python 3.X (наряду со многими другими), многие пользователи Python 2.X в наши дни по-прежнему благополучно избегают такой проблемы.

Каковы технические превосходства Python?

Естественно, это вопрос разработчика. Если опыт программирования у вас пока отсутствует, то материал в нескольких последующих разделах может сбивать с толку — не переживайте, далее мы в книге детально исследуем все необходимые вопросы. Разработчики найдут ниже краткое введение в основные технические характеристики Python.

Он объектно-ориентированный и функциональный

Python — объектно-ориентированный язык во всех отношениях. Его модель классов поддерживает такие расширенные понятия, как полиморфизм, перегрузка операций и множественное наследование, но в контексте простого синтаксиса и типизации Python применять ООП необыкновенно легко. Между прочим, если вы не понимаете упомянутые термины, то обнаружите, что их гораздо легче изучить с помощью Python, чем любого другого доступного языка ООП.

Помимо того, что объектно-ориентированная природа служит мощным механизмом структурирования и многократного использования кода, она делает его идеальным в качестве инструмента написания сценариев для других языков ООП. Например, посредством подходящего связующего кода программы Python могут создавать подклассы (специализировать) классов, реализованных на C++, Java и C#.

Не менее значимо и то, что ООП — лишь возможный *вариант* в Python; вы можете многое добиться, одновременно не становясь гуру ООП. Во многом подобно C++ язык Python поддерживает режимы процедурного и объектно-ориентированного программирования. Его объектно-ориентированные инструменты могут применяться, если это допускают ограничения, что особенно полезно в тактических режимах разработки, где стадии проектирования отсутствуют.

В дополнение к первоначальным *процедурной* (основанной на операторах) и *объектно-ориентированной* (основанной на классах) парадигмам за последние годы Python обзавелся встроенной поддержкой для *функционального программирования* — набора, который включает генераторы, включения, замыкания, отображения, декораторы, анонимные лямбда-функции и объекты функций первого класса. Они могут служить как дополнением, так и альтернативой инструментам ООП.

Он бесплатный

Python используется и распространяется совершенно бесплатно. Подобно другому ПО с открытым кодом, такому как Tcl, Perl, Linux и Apache, вы можете бесплатно получить через Интернет исходный код полной системы Python. Не существует никаких ограничений на его копирование, встраивание в системы или поставку с собственными продуктами. На самом деле вы даже можете продавать исходный код Python, если у вас к этому склонность.

Но не воспринимайте неправильно: “бесплатный” вовсе не означает “неподдерживаемый”. Наоборот, онлайновое сообщество Python реагирует на запросы пользователей со скоростью, которой бы позавидовали многие службы технической поддержки коммерческого ПО. Кроме того, поскольку Python сопровождается полным исходным кодом, он расширяет возможности разработчиков, приводя к созданию крупной команды экспертов по реализации. Хотя изучение либо изменение реализации языка программирования вызывает восторг далеко не у каждого, приятно осознавать, что подобное возможно, если в этом возникнет потребность. Вы не зависите от прихотей коммерческого поставщика, потому что в вашем распоряжении находится исчерпывающая документация — *исходный код* — как последнее средство.

Ранее упоминалось, что разработка Python осуществляется сообществом, которое в значительной степени координирует свои усилия через Интернет. В сообщество входит первоначальный создатель Python, *Гвидо ван Россум*, официально нареченный *Великодушным пожизненным диктатором Python* (*Benevolent Dictator for Life — BDFL*), и второй состав, исчисляемый тысячами. Изменения языка должны следовать формальной процедуре расширения и тщательно исследуются другими разработчиками и BDFL. Это делает Python консервативным в отношении изменений по сравнению с рядом других языков и систем. Несмотря на то что разделение Python 3.X/2.X обоснованно и преднамеренно разорвало связь с данной традицией, в целом она остается справедливой внутри каждой линейки Python.

Он переносимый

Стандартная реализация Python написана на переносимом языке ANSI C; она компилируется и запускается практически на любой крупной платформе, применяемой в настоящее время. Например, программы Python выполняются в наши дни на всем, начиная с персональных цифровых секретарей и заканчивая суперкомпьютерами. Вот неполный список того, где доступен Python:

- системы Linux и Unix;
- Microsoft Windows (все современные разновидности);
- Mac OS (OS X и Classic);
- BeOS, OS/2, VMS и QNX;
- системы реального времени, такие как VxWorks;
- суперкомпьютеры Cray и майнфреймы IBM;
- персональные цифровые секретари, управляемые Palm OS, PocketPC и Linux;
- мобильные телефоны, функционирующие под управлением Symbian OS и Windows Mobile;
- игровые приставки и устройства iPod;
- планшеты и смартфоны, управляемые Google Android и Apple iOS;
- многие другие.

Подобно самому интерпретатору языка стандартные библиотечные модули, которые поставляются вместе с Python, реализованы так, чтобы быть максимально переносимыми между границами платформ. Более того, программы Python автоматически компилируются в переносимый байт-код, который выполняется в неизменном виде на любой платформе с установленной совместимой версией Python (более подробно об этом пойдет речь в следующей главе).

Сказанное означает, что программы Python, использующие основной набор языка и стандартные библиотеки, выполняются без изменений в Linux, Windows и большинстве других систем с интерпретатором Python. Большинство переносимых версий Python также содержат расширения, специфичные для платформ (скажем, поддержка COM в Windows), но основной набор языка Python и библиотеки работают одинаково повсюду. Как упоминалось ранее, Python также включает интерфейс к инструментальному набору Tk GUI API под названием tkinter (Tkinter в Python 2.X), который позволяет реализовывать в программах Python полнофункциональные графические пользовательские интерфейсы, функционирующие без каких-либо изменений на всех важных настольных графических plataформах.

Он мощный

С точки зрения возможностей Python является чем-то вроде гибрида. Имеющийся инструментальный набор помещает его между традиционными языками для написания сценариев (наподобие Tcl, Scheme и Perl) и языками для разработки систем (такими как C, C++ и Java). Python обеспечивает всю простоту и легкость использования языка для написания сценариев наряду с более развитыми инструментами для разработки ПО, обычно обнаруживаемыми в компилируемых языках. В отличие от ряда языков написания сценариев такое сочетание делает Python удобным для разработки крупномасштабных проектов. В качестве предварительного представления ниже кратко описано то, что вы найдете в инструментальном наборе Python.

Динамическая типизация

Python отслеживает виды объектов, которые ваша программа применяет во время выполнения; он не требует сложных объявлений типов и размеров в коде. На самом деле, как вы увидите в главе 6, в Python вообще нет такого понятия, как объявление типа или переменной. Поскольку код Python не ограничивается типами данных, он также обычно применим к целому диапазону объектов.

Автоматическое управление памятью

Python автоматически выделяет память под объекты и возвращает ее обратно (“собирает мусор”), когда объекты больше не используются, и большинство объектов могут расти и сокращаться в размерах по требованию. Как вы узнаете, Python отслеживает низкоуровневые детали, связанные с памятью, так что вам делать это не придется.

Поддержка “программирования в большом”

Для построения более крупных систем в Python предусмотрены инструменты, подобные модулям, классам и исключениям. Инструменты такого рода дают возможность организовывать системы в виде компонентов, применять ООП для многократного использования и настройки кода, а также элегантно обрабатывать события и ошибки. Описанные ранее инструменты функционального программирования Python предлагают дополнительные способы для достижения множества тех же самых целей.

Встроенные типы объектов

Python предоставляет часто используемые структуры данных вроде списков, словарей и строк как собственные части языка; вы увидите, что они являются гибкими и легкими в употреблении. Скажем, встроенные объекты могут рассти и сокращаться в размерах по требованию, произвольно вкладываться друг в друга для представления сложной информации и т.д.

Встроенные инструменты

Для обработки всех типов объектов в Python предусмотрены мощные стандартные операции, в числе которых конкатенация (объединение коллекций), нарезание (извлечение разделов), сортировка, отображение и т.д.

Библиотечные утилиты

Для более специфических задач в Python также имеется большой набор готовых библиотечных инструментов, которые поддерживают все, начиная с сопоставления с регулярными выражениями и заканчивая работой с сетью. После того, как вы изучите сам язык, библиотечные инструменты Python будут тем местом, где происходит большинство действий прикладного уровня.

Сторонние утилиты

Из-за открытости кода Python разработчикам рекомендуется предоставлять готовые инструменты, поддерживающие задачи помимо тех, которые поддерживаются встроенными утилитами; в веб-сети вы обнаружите поддержку для COM, обработки изображений, численного программирования, XML, доступа к базам данных и многое другое.

Несмотря на наличие массы инструментов в Python, он сохраняет удивительно простой синтаксис и структуру. Результатом оказывается мощный инструмент программирования со всеми удобствами языка написания сценариев.

Он смешиваемый

Программы Python могут быть легко “связаны” с компонентами, написанными на других языках, самыми разнообразными способами. Например, API-интерфейс C, имеющийся в Python, позволяет программам C гибким образом вызывать и быть вызванными программами Python. В результате вы можете по мере необходимости добавлять функциональность к системе Python и применять программы Python внутри других сред или систем.

Смешивание Python с библиотеками, написанными на языке C или C++, например, делает его легким в использовании интерфейсным языком и инструментом настройки. Как упоминалось ранее, это также делает Python хорошим вариантом для быстрого прототипирования — системы могут быть реализованы сначала на Python, чтобы задействовать в своих интересах обеспечиваемую им скорость разработки, и позже переноситься на C для поставки, по одной порции за раз, согласно требованиям к производительности.

Он относительно прост в использовании

В сравнении с альтернативами вроде C++, Java и C# для большинства наблюдателей программирование на Python выглядит удивительно простым. Чтобы выполнить про-

грамму Python, вы просто набираете ее и запускаете. Промежуточные шаги компиляции и связывания, как в языках С и С++, не требуются. Python выполняет программы тотчас же, что сделано для поддержки интерактивного программирования и ускоренного цикла после внесения изменений в программу — во многих случаях вы можете наблюдать эффект от изменения программы почти так же быстро, как только способны набирать его.

Конечно, цикл разработки — лишь один аспект легкости использования Python. Он также обеспечивает преднамеренно простой синтаксис и мощные встроенные инструменты. На самом деле некоторые заходят настолько далеко, что называют Python *исполняемым псевдокодом*. Из-за устранения большей части сложности в других инструментах программы Python проще, меньше и гибче своих эквивалентов, написанных на других популярных языках.

Он относительно прост в изучении

Это подводит нас к цели настоящей книги: базовый язык Python необыкновенно легко изучать, особенно если сравнивать с другими широко применяемыми языками программирования. Фактически, если вы — опытный программист, тогда можете расчитывать, что вы будете писать небольшие программы на Python через считанные дни. Чтобы уловить некоторые ограниченные части языка, вам и вовсе понадобятся считанные часы — хотя вы не должны ожидать, что станете экспертом настолько быстро (вопреки тому, что вы могли слышать от сотрудников отдела маркетинга!).

Естественно, освоение любой темы, настолько значительной, как современный язык Python, не будет простым, и данной задаче посвящен остаток книги. Но надлежащие вложения, необходимые для овладения Python, того стоят — в конечном счете вы получите навыки программирования, которые применимы практически к каждой предметной области, связанной с компьютерами. Кроме того, большинство находит кривую обучения Python намного более пологой, чем у других инструментов программирования.

Это хорошая новость для профессиональных разработчиков, стремящихся изучить язык с целью использования на работе, а также для конечных пользователей систем, которые открывают доступ к уровню Python для настройки или управления. В наши дни многие системы полагаются на тот факт, что конечные пользователи могут изучить язык Python в достаточной степени, чтобы подгонять код настройки на Python по месту эксплуатации с небольшой поддержкой или вовсе без нее. Более того, Python породил крупную группу пользователей, программирующих ради удовольствия, а не карьеры, которые могут не нуждаться в навыках разработки полномасштабного ПО. Хотя Python располагает расширенными инструментами программирования, базовые основы языка все равно будут касаться относительно простыми как начинающим, так и гуру.

Он назван в честь группы “Монти Пайтон”

Хорошо, это не совсем техническое превосходство, но выглядит как неожиданно хорошо хранимый секрет в мире Python, который я хочу раскрыть авансом. Вопреки всем рептилиям, изображаемым на книгах и значках Python, правда заключается в том, что Python назван в честь британской комик-группы “Монти Пайтон” (Monty Python) — создателей комедийного сериала “Летающий цирк Монти Пайтона”, вышедшего на Би-Би-Си в 1970-х годах, и позже ряда полнометражных фильмов, в числе которых “Монти Пайтон и Священный Грааль”, по-прежнему популярный и сегодня. Первоначальный создатель Python был почитателем “Монти Пайтон”, как и многие разработчики ПО (похоже, между этими двумя областями есть некая симметрия...).

Такое наследие неизбежно привносит юмористические черты в примеры кода Python. Например, традиционные имена “foo” и “bar” для обобщенных переменных в мире Python превратились в “spam” и “eggs”. Встречающиеся временами “Brian”, “ni” и “shrubbery” также обязаны своим появлением соответствующим тезкам. Не осталось без влияния даже сообщество Python в целом: некоторые мероприятия на конференциях по Python регулярно объявляются как “испанская инквизиция”.

Разумеется, все это забавно, если вы знакомы с шоу, но малозначительно, если нет. Вы не обязаны знать работы комик-группы “Монти Пайтон”, чтобы понимать примеры, в которых заимствуются ссылки на них, включая приведенные в настоящей книге, но, во всяком случае, теперь вам известно, откуда все происходит.

Как Python соотносится с языком X?

Наконец, чтобы поместить Python в контекст того, что вы уже возможно знаете, люди иногда сравнивают его с такими языками, как Perl, Tcl и Java. В разделе подводится итог по общему согласию в данном вопросе.

Я хочу заранее отметить, что не являюсь любителем выигрывать, унизяя достоинство соперников — в длительной перспективе это не работает, и цель здесь вовсе не в том. Кроме того, это не игра с нулевой ставкой — большинство программистов за время своей профессиональной деятельности будут пользоваться многими языками. Тем не менее, инструменты программирования предоставляют выбор и компромиссы, заслуживающие рассмотрения. В конце концов, если бы Python не предлагал чего-то сверх имеющихся альтернатив, то его никогда и не применяли бы в первую очередь.

Ранее мы говорили о компромиссах, связанных с производительностью, так что теперь сконцентрируемся на функциональности. Хотя другие языки также являются полезными инструментами, чтобы их знать и использовать, многие люди считают, что Python обладает следующими характеристиками.

- Он мощнее *Tcl*. Сильная поддержка Python “программирования в большом” делает его пригодным для разработки крупных систем, а его библиотека прикладных инструментов более обширна.
- Он читабельнее *Perl*. Язык Python имеет ясный синтаксис и простую, согласованную структуру, что в свою очередь увеличивает возможность многократного применения и удобство сопровождения кода Python, а также помогает сократить число ошибок в программах.
- Он проще и легче в использовании, чем *Java* и *C#*. Python — язык написания сценариев, но Java и C# унаследовали большую часть сложности и синтаксиса у языков ООП вроде C++.
- Он проще и легче в применении, чем C++. Код Python проще эквивалентного кода C++ и часто меньше на одну третью или пятую часть, хотя как язык написания сценариев Python временами исполняет различные роли.
- Он проще и выше по уровню, чем C. Отделение Python от лежащей в основе аппаратной архитектуры делает код менее сложным, лучше структурированным и более доступным, чем код на языке C, прародителе C++.
- Он более мощный, универсальный и межплатформенный, чем *Visual Basic*. Язык Python богаче и используется более широко, а его природа открытости кода означает, что он не контролируется одной компанией.

- Он более читабельный и универсальный, чем *PHP*. Python используется и при конструировании веб-сайтов, но он также применим почти ко всем остальным предметным областям, связанным с компьютерами, от роботов до анимации и игр.
- Он более мощный и универсальный, чем *JavaScript*. Python имеет больший инструментальный набор и не настолько тесно привязан к веб-разработке. Он также используется для научного моделирования, оснащения инструментами и т.д.
- Он более читабельный и установившийся, чем *Ruby*. Синтаксис Python менее беспорядочный, особенно в нетривиальном коде, и его ООП совершенно не обязательно для пользователей и проектов, к которым оно может быть неприменимо.
- Он более зрелый и широко ориентированный, чем *Lua*. Более крупный набор возможностей и всесторонняя библиотечная поддержка Python обеспечивает ему более широкие границы использования по сравнению с *Lua*, встраиваемым “связующим” языком наподобие *Tcl*.
- Он менее экзотичный, чем *Smalltalk*, *Lisp* и *Prolog*. Python является динамической разновидностью указанных языков, но имеет традиционный синтаксис, понятный разработчикам и конечным пользователям настраиваемых систем.

В особенности для программ, которые делают нечто большее, чем просмотр текстовых файлов, и которые в будущем, возможно, придется читать другим разработчикам (или вам!), многие люди считают, что Python отвечает всем требованиям лучше других языков написания сценариев или программирования, доступных на сегодняшний день. Кроме того, если только вашему приложению не требуется пиковая производительность, то Python зачастую оказывается реальной альтернативой языкам разработки систем, таким как C, C++ и Java: код Python нередко может добиваться тех же целей, но будет гораздо легче в написании, отладке и сопровождении.

Конечно, с учетом того, что еще с 1992 года ваш автор был ярым сторонником Python с удостоверением в кармане, воспринимайте приведенные выше комментарии как считаете нужным (и полезность доводов сторонников других языков может произвольно варьироваться). Однако они отражают общий опыт многих разработчиков, потративших время на исследование того, что предлагает Python.

Резюме

На этом “хвалебная” часть книги завершена. В главе вы узнали причины, по которым люди выбирают Python для решения имеющихся задач программирования. Вы также увидели, как он применяется, и ознакомились с презентативной выборкой тех, кто его использует в наши дни. Тем не менее, моя цель – обучать Python, а не продавать его. Язык лучше всего оценивать, наблюдая его в действии, и потому остаток книги сосредоточен полностью на деталях языка, которые здесь не были раскрыты.

В последующих двух главах начинается формальное представление языка. В них вы узнаете способы запуска программ Python, быстро взглянете на модель выполнения на базе байт-кода и ознакомитесь с основами файлов модулей, предназначенных для хранения кода. Целью будет снабжение вас информацией, достаточной для выполнения примеров и упражнений в оставшихся главах книги. По существу настоящее программирование начнется лишь в главе 4, но до того необходимо должным образом подготовиться.

Проверьте свои знания: контрольные вопросы

В этом издании книги каждая глава будет завершаться перечнем контрольных вопросов, касающихся пройденного материала, что поможет закрепить знание основных концепций. Сразу же за контрольными вопросами приводятся ответы на них, и вы можете прочитать их после того, как попробуете справиться с вопросами самостоятельно, поскольку иногда они дают полезный контекст.

Помимо контрольных вопросов, приводимых в конце глав, в конце каждой части вы найдете лабораторные *упражнения*, призванные помочь вам самостоятельно начать написание кода Python. Итак, вот первый набор контрольных вопросов. Удачи и обязательно при необходимости возвращайтесь к материалам данной главы.

1. Каковы шесть основных причин, по которым люди выбирают Python?
2. Назовите четыре видных компании или организации, которые применяют Python в настоящее время.
3. Почему вы можете не захотеть использовать Python в приложении?
4. Что можно делать с помощью Python?
5. В чем важность команды `import this` в Python?
6. Почему слово “spam” обнаруживается в настолько многих примерах кода Python книгах и веб-сети?
7. Какой ваш любимый цвет?

Проверьте свои знания: ответы

Справились с вопросами? Ниже приведены мои ответы, хотя у некоторых контрольных вопросов может быть несколько решений. Даже если вы уверены в своих ответах, я призываю вас взглянуть на мои варианты, чтобы ознакомиться с дополнительными обстоятельствами. Если любые из ответов непонятны, тогда проработайте материал главы еще раз.

1. Качество ПО, продуктивность труда разработчиков, переносимость программ, библиотеки поддержки, интеграция компонентов и просто наслаждение. Из всего перечисленного качество и продуктивность представляются главными причинами, по которым люди выбирают Python.
2. Google, Industrial Light & Magic, CCP Games, Jet Propulsion Labs, Maya, ESRI и многие другие. Почти каждая организация, занимающаяся разработкой ПО, применяет Python в каком-нибудь виде, либо для долгосрочной стратегической разработки продуктов, либо для решения краткосрочных тактических задач, подобных тестированию и администрированию систем.
3. Главным недостатком Python является производительность: он не будет выполняться насколько же быстро, как полностью компилируемые языки вроде C и C++. С другой стороны, его производительность вполне достаточна для большинства приложений, и типичный код Python так или иначе выполняется со скоростью, близкой к скорости C, поскольку он обращается к связанному коду C в интерпретаторе. На случай, когда скорость критически важна, доступны скомпилированные расширения для частей приложения, отвечающих за перемалывание чисел.

4. Язык Python можно использовать практически для всего того, что можно делать с помощью компьютера, начиная с разработки веб-сайтов и игр и заканчивая управлением роботами и космическими аппаратами.
5. Она упоминалась в сноске: команда `import this` вызывает “информацию-сюрприз”, скрытую в Python, которая отображает принципы проектирования, положенные в основу языка. В следующей главе вы узнаете, как запускать ее.
6. Слово “spam” взято из известной пародии комик-группы “Монти Пайтон”, где люди пытаются заказать еду в кафетерии, но их заглушает хор викингов, поющих о мясных консервах марки “Spam”. И да, это также распространенное имя переменной в сценариях Python...
7. Синий. Нет, желтый! (См. предыдущий ответ.)

Python – это инженерия, а не искусство

Когда Python впервые появился на сцене ПО в начале 1990-х годов, он породил то, что сейчас расценивается как классический конфликт между его сторонниками и защитниками еще одного популярного языка написания сценариев, Perl. Лично я считаю, что интерес к полемике утрачен и в наши дни она неуместна – разработчики достаточно сообразительны, чтобы самостоятельно сделать выводы. Однако это один из самых распространенных вопросов, который мне задают в ходе обучения, и он подчеркивает одну из главных причин выбора Python, а потому вполне уместно сказать здесь о нем несколько слов.

Краткая история такова: *вы можете делать на Python все то, что можно сделать на Perl, но вы в состоянии прочитать свой код после того, как написали его.* Вот и все – их предметные области в значительной степени перекрываются, но язык Python больше сосредоточен на выпуске читабельного кода. Для многих улучшенная читабельность Python превращается в большие возможности многократного применения и удобства сопровождения, делая Python более подходящим вариантом для программ, которые не относятся к тем, что пишутся один раз и затем отбрасываются. Код Perl легко писать, но может быть трудно читать. Учитывая, что большинство программ имеют срок службы, гораздо более длительный, чем их начальное создание, многие видят Python как более эффективный инструмент.

Несколько более длинная история отражает происхождение проектировщиков двух языков. Python был создан математиком по образованию, который, кажется, естественным образом синтезировал ортогональный язык с высокой степенью единообразия и согласованности. Perl был рожден лингвистом, который создавал инструмент программирования близким к естественному языку, с его чувствительностью к контексту и большой изменчивостью. Как гласит хорошо известный девиз Perl, *есть более одного способа сделать это.* Исходя из такого образа мыслей, язык Perl и сообщество его пользователей исторически поощряли неограниченную свободу выражения при написании кода. Код Perl, написанный одним, может совершенно отличаться от кода Perl, написанного другим. На самом деле написание уникального и мудреного кода часто является предметом гордости у пользователей Perl.

Тем не менее, как должен подтвердить любой, кто занимался сопровождением сколько-нибудь значительного кода, *свобода выражения замечательна для искусства, но плоха для инженерии.* В инженерии нам необходим минимальный набор возможностей и предсказуемость. В инженерии свобода выражения может привести к кошмару при сопровождении. Как уже несколько пользователей Perl сообщили мне по секрету, в результате слишком большой свободы зачастую появляется код, который легче переписать с нуля, чем модифицировать. Ситуация явно далека от идеала.

Подумайте о следующем: когда люди создают картину или скульптуру, то делают это в основном для себя; шанс на то, что кто-то другой изменит их работу в будущем, во внимание не принимается. В том и заключается критическое отличие между искусством и инженерией. Когда люди пишут ПО, они не делают это для себя. В сущности, они даже не пишут в первую очередь для компьютера. Наоборот, хорошие программисты знают, что код пишется для следующего человека, который должен прочитать его, чтобы сопровождать либо использовать повторно. Если этот человек не сможет понять код, то он практически бесполезен в реальном сценарии разработки. Другими словами, программирование не должно быть заумным и неясным; *оно касается того, насколько четко ваша программа сообщает о своей цели.*

Концентрация на читабельности – тот аспект, который согласно мнению многих наиболее ярко отличает Python от других языков написания сценариев. Поскольку синтаксическая модель Python едва ли не *заставляет* создавать читабельный код, программы Python лучше приспособливаются к полному циклу разработки ПО. А учитывая, что в Python делается особый акцент на таких идеях, как ограниченное взаимодействие, единообразие кода и согласованность функциональных средств, он больше благоприятствует получению кода, который еще долго может применяться после того, как он был первоначально написан.

В общем случае концентрация Python на *качестве кода* сама по себе повышает производительность труда программистов, а также степень их удовлетворенности. Конечно, программисты на Python также могут быть очень креативными, и как мы увидим, для ряда задач язык предлагает множество решений – временами даже больше, чем должен на сегодняшний день (проблема, с которой мы столкнемся в этой книге). Фактически данную врезку можно воспринимать как *поучительную историю*: качество оказывается хрупким состоянием, которое одинаково сильно зависит и от людей, и от технологии. Python исторически поощрял хорошую инженерию способами, которые у других языков написания сценариев часто отсутствовали, но остаток истории о качестве вы допишете сами.

Во всяком случае, так выглядят некоторые выработанные общими усилиями мнения многих людей, принявших Python. Разумеется, вы должны оценивать такие утверждения применительно к себе, изучая то, что способен предложить Python. Давайте перейдем к следующей главе, которая поможет вам начать работу.

Как Python выполняет программы

В этой и следующей главах кратко рассматривается выполнение программ — как вы запускаете код, и каким образом его Python выполняет. В главе вы узнаете, как интерпретатор Python выполняет программы вообще. Затем в главе 3 будет показано, каким образом готовить и запускать собственные программы.

По существу детали запуска специфичны для платформы, и определенные материалы в текущей и следующей главе могут быть неприменимыми к платформе, на которой вы работаете. Более опытные читатели должны спокойно пропускать то, что не имеет к ним отношения. Аналогично читатели, которые уже пользовались в прошлом похожими инструментами и предпочитают быстрее перейти к изучению языка, могут оставить эти главы “на будущее”. Остальным читателям рекомендуется бегло взглянуть на то, как Python будет выполнять код, прежде чем учиться писать его.

Введение в интерпретатор Python

До сих пор я говорил о Python по большей части как о языке программирования. Но, как реализовано в текущий момент, Python представляет собой также программный пакет, называемый *интерпретатором*. Интерпретатор — это разновидность программы, которая выполняет другие программы. Когда вы пишете программу на Python, интерпретатор Python читает ее и приводит в исполнение содержащиеся в ней инструкции. Фактически интерпретатор является уровнем программной логики между вашим кодом и оборудованием компьютера.

В результате установки Python на машине создается несколько компонентов — минимум интерпретатор и библиотека поддержки. В зависимости от способа применения интерпретатор Python может принимать форму исполняемой программы или набора библиотек, связанных с другой программой. В зависимости от используемой разновидности Python сам интерпретатор может быть реализован в виде программы C, набора классов Java или чего-нибудь другого. Независимо от формы код Python, который вы пишете, всегда должен выполняться интерпретатором. Чтобы сделать это возможным, вы обязаны установить на своем компьютере интерпретатор Python.

Детали установки Python изменяются от платформы к платформе и более подробно рассматриваются в приложении А второго тома. Ниже приведены краткие замечания.

- Пользователи Windows загружают и запускают самоустанавливающийся исполняемый файл, который помещает Python на их компьютеры. Понадобится просто дважды щелкнуть на имени файла и утвердительно отвечать на все последующие запросы.

- Пользователи Linux и Mac OS X, возможно, уже располагают заранее установленной копией Python на своих компьютерах – в настоящее время это стандартный компонент для их платформ.
- Некоторые пользователи Linux и Mac OS X (а также большинство пользователей Unix) компилируют Python из пакета с полным исходным кодом.
- Пользователи Linux также могут найти файлы RPM, а пользователи Mac OS X – разнообразные установочные пакеты, специфичные для Macs.
- На других plataформах предусмотрены свои приемы установки. Например, Python доступен на мобильных телефонах, планшетах, игровых приставках и устройствах iPod, но детали установки варьируются в широких пределах.

Сам Python можно загрузить из основного веб-сайта <http://www.python.org>. Он также доступен через многие другие каналы распространения. Не забывайте перед установкой проверять, присутствует ли уже копия Python в системе. Если вы работаете под управлением Windows 7 или предшествующей версии, то обычно найдете Python в меню Start (Пуск), как показано на рис. 2.1; мы обсудим пункты меню в следующей главе. На компьютерах Unix и Linux, скорее всего, Python находится в дереве каталогов /usr.

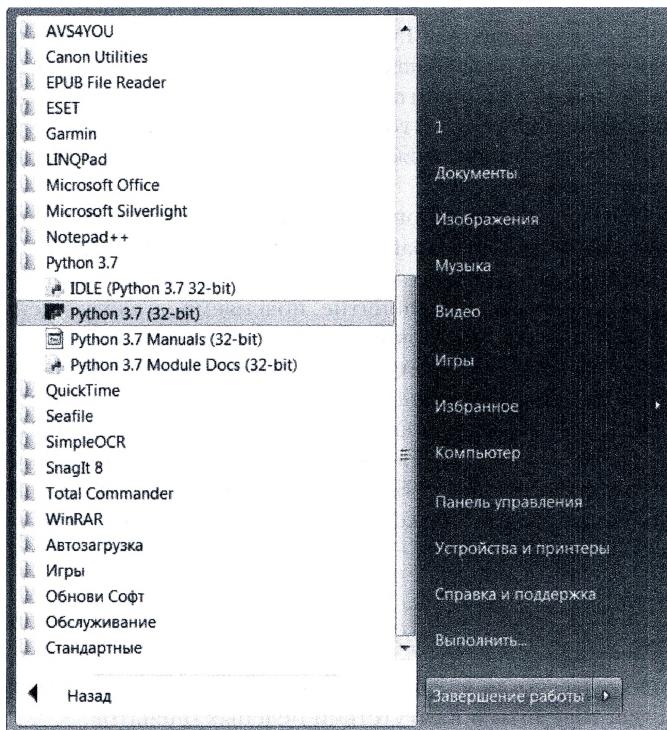


Рис. 2.1. В случае работы под управлением Windows 7 или предшествующей версии Python доступен через меню Start. Структура может изменяться в зависимости от выпуска, но IDLE запускает графический пользовательский интерфейс разработки, а Python – простой интерактивный сеанс. Также здесь доступны стандартные руководства и документация PyDoc (Module Docs). Указания для Windows 8 и других платформ приведены в главе 3 и приложении А (том 2)

Поскольку детали установки сильно зависят от платформы, мы не будем приводить здесь их описание. Дополнительные сведения о процессе установки представлены в приложении А второго тома. Для целей настоящей и следующей глав мы будем полагать, что Python установлен и готов к работе.

Выполнение программ

То, что конкретно стоит за написанием и выполнением сценария Python, зависит от того, с чьей точки зрения рассматривать эти задачи – программиста или интерпретатора Python. Обе точки зрения выражают важные взгляды, касающиеся программирования на Python.

Точка зрения программиста

В своей простейшей форме программа на Python представляет собой всего лишь текстовый файл, содержащий операторы Python. Например, в показанном ниже содержимом файла по имени `script0.py` находится простейший сценарий Python, какой только мне удалось придумать, но он является полнофункциональной программой Python:

```
print('hello world')
print(2 ** 100)
```

Файл `script0.py` содержит два Python-оператора `print`, которые выводят в выходной поток строку (текст в кавычках) и результат числового выражения (2 в степени 100). Пока не беспокойтесь о синтаксисе приведенного кода – в данной главе нас интересует только приведение его в готовность к запуску. Я объясню работу оператора `print` и то, почему в Python вы можете возвести 2 в степень 100 без переполнения, в последующих частях книги.

Вы можете создать такой файл с операторами в любом текстовом редакторе, который вам нравится. По соглашению файлам с программами Python назначаются имена, оканчивающиеся на `.py`; формально такая схема именования обязательна только для файлов, которые “импортируются” (понятие, поясняемое в следующей главе), но ради согласованности большинство файлов Python имеют имена `.py`.

После набора всех операторов в текстовом файле вы должны сообщить интерпретатору Python о необходимости выполнить файл – что просто означает запуск всех операторов в файле сначала до конца, друг за другом. Как объясняется в следующей главе, вы можете запускать файлы программ Python через строки командной оболочки, путем щелчка на их значках, из IDE-сред и с помощью других стандартных приемов. Если выполнение файла проходит гладко, тогда вы увидите результаты двух операторов `print` в каком-то месте – по умолчанию в том же окне, где запускалась программа:

```
hello world
1267650600228229401496703205376
```

Бот что происходит, когда я запускаю этот сценарий в окне командной строки Windows, чтобы удостовериться в отсутствии нелепых опечаток:

```
C:\code> python script0.py
hello world
1267650600228229401496703205376
```

Ищите в главе 3 полное изложение процесса, особенно если вы новичок в программировании; там вы узнаете мельчайшие подробности написания и запуска про-

грамм. Здесь же мы просто выполняем сценарий Python, который выводит строку и число. Вряд ли мы получим какую-нибудь награду по программированию за этот код, но его достаточно, чтобы уловить основы выполнения программ.

Точка зрения Python

Краткое описание в предыдущем разделе довольно стандартно для языков написания сценариев и обычно это все, что нужно знать большинству программистов на Python. Вы помещаете код в текстовые файлы и запускаете их через интерпретатор. Однако когда вы сообщаете Python о запуске, “за кулисами” происходит нечто большее. Хотя знание внутреннего устройства Python не является обязательным требованием для программирования на Python, базовое понимание структуры исполняющей среды Python может помочь уловить общую картину выполнения программ.

Когда вы инструктируете Python относительно выполнения сценария, то перед тем, как ваш код начнет перемалывание чисел, Python предпринимает несколько шагов. В частности, первым делом он осуществляет компиляцию сценария в то, что называется “байт-кодом”, и направляет его так называемой “виртуальной машине”.

Компиляция в байт-код

После запуска программы Python внутренне и почти полностью скрыто от вас сначала компилирует ваш *исходный код* (операторы внутри файла) в формат, известный как *байт-код*. Компиляция является просто шагом трансляции, а байт-код – это низкоуровневое и независимое от платформы представление исходного кода. Грубо говоря, Python транслирует каждый оператор исходного кода в группу инструкций байт-кода, разбивая их на отдельные шаги. Такая трансляция в байт-код происходит со скоростью выполнения – байт-код способен выполняться быстрее, чем первоначальные операторы исходного кода из текстового файла.

Вы наверняка заметили, что в предыдущем абзаце было указано, что компиляция *почти* полностью скрыта от вас. Если процесс Python имеет доступ по записи на вашем компьютере, тогда он будет сохранять программы в файлах с расширением .рус (означает скомпилированный (compiled) исходный файл .ру). До выхода версии Python 3.2 эти файлы появлялись на компьютере после запуска нескольких программ там, где находились соответствующие файлы исходного кода, т.е. в *тех же самых* каталогах. Например, после импортирования script.rus появлялся файл script.rus.

В Python 3.2 и последующих версиях файлы байт-кода .rus взамен сохраняются в подкаталоге по имени __ruscache__, расположенному в каталоге, где находятся файлы исходного кода, и их имена идентифицируют версию Python, в которой они создавались (скажем, script.cpython-33.rus). Новый подкаталог __ruscache__ помогает избежать беспорядка, а новое соглашение об именовании файлов байт-кода предотвращает переписывание сохраненного байт-кода разными версиями Python, которые могут быть установлены на одном компьютере. Мы исследуем модели файлов байт-кода более подробно в главе 22, хотя они создаются автоматически, не имеют отношения к большинству программ Python и могут варьироваться среди альтернативных реализаций Python, описываемых далее.

В обеих моделях Python сохраняет байт-код подобного рода в качестве оптимизации скорости начального загрузки. Во время следующего запуска программы Python загрузит файлы .rus и пропустит шаг компиляции при условии, что вы не изменили исходный код после того, как байт-код был сохранен, и не выполняете версию Python, отличающуюся от той, которая создавала байт-код. Вот как все работает.

- **Изменения исходного кода.** Python автоматически проверяет отметки времени последней модификации для файлов исходного кода и байт-кода, чтобы выяснить, когда они должны быть перекомпилированы — если вы отредактируете и повторно сохраните исходный код, то байт-код будет автоматически создан заново при следующем запуске программы.
- **Версии Python.** Импортирования также проверяются для выяснения, подлежит ли перекомпиляции файл из-за того, что он был создан другой версией Python, используя либо “магический” номер версии в самом файле байт-кода для Python 3.2 и предшествующих версий, либо информацию, присутствующую в имени файла байт-кода для Python 3.2 и последующих версий.

В результате внесение изменений в исходный код и применение отличающихся номеров версий Python будет инициировать создание нового файла байт-кода. Если Python не имеет возможности записывать файлы байт-кода на вашем компьютере, то программа все равно будет работать — байт-код генерируется в памяти и просто отбрасывается по завершении выполнения программы. Тем не менее, поскольку файлы .рус ускоряют загрузку, для крупных программ имеет смысл обеспечить возможность записи файлов байт-кода. Файлы байт-кода представляют собой также один из способов поставки программ Python — они будут выполняться, даже если все, что удается найти — это файлы .рус, а файлы исходного кода .ру отсутствуют. (Еще один вариант поставки описан в разделе “Фиксированные двоичные файлы” далее в главе.)

Наконец, имейте в виду, что байт-код сохраняется лишь для тех файлов, которые *импортируются*, но не для файлов верхнего уровня программы, выполняемых только как сценарии (строго говоря, речь идет об оптимизации импорта). Мы исследуем основы импорта в главе 3, а более подробно рассмотрим его в части V. Кроме того, заданный файл импортируется (и возможно компилируется) лишь *один раз* на запуск программы, а байт-код никогда не сохраняется для кода, набираемого в строке *интерактивной подсказки* — режиме программирования, о котором вы узнаете в главе 3.

Виртуальная машина Python (PVM)

После того, как программа скомпилирована в байт-код (или байт-код был загружен из существующих файлов .рус), она отправляется на выполнение тому, что в общем случае известно как *виртуальная машина Python* (Python Virtual Machine — PVM). Машина PVM производит более глубокое впечатление, чем есть на самом деле; в действительности она не является отдельной программой и сама по себе не требует установки. По сути, машина PVM — всего лишь большой закодированный цикл, который проходит по инструкциям вашего байт-кода, друг за другом, чтобы выполнить их операции. Машина PVM — это исполняющий механизм Python; она всегда присутствует в виде части системы Python и представляет собой компонент, который по-настоящему выполняет ваши сценарии. Формально она является просто последним шагом того, что называется “*интерпретатор Python*”.

На рис. 2.2 иллюстрируется структура описанного здесь исполняющего механизма. Имейте в виду, что вся эта сложность намеренно скрыта от программистов на Python. Компиляция в байт-код происходит автоматически, а машина PVM — лишь часть системы Python, установленной на компьютере. Опять-таки программисты просто набирают код и запускают файлы с операторами, а Python обеспечивает возможность их выполнения.



Рис. 2.2. Традиционная модель выполнения Python: набираемый вами исходный код транслируется в байт-код, который затем выполняется виртуальной машиной Python.

Исходный код автоматически компилируется и впоследствии интерпретируется

Последствия в плане производительности

Читатели с опытом программирования на полностью компилируемых языках, таких как С и С++, могут отметить несколько отличий в модели Python. Во-первых, в работе Python обычно отсутствует шаг построения или “сборки”: код выполняется немедленно после того, как был набран. Во-вторых, байт-код Python не является двоичным машинным кодом (например, инструкциями для процессора Intel или ARM). Байт-код – это представление, специфичное для Python.

Вот почему определенный код Python может выполняться не настолько быстро, как код С или С++ (см. главу 1): циклу PVM, а не центральному процессору, необходимо интерпретировать байт-код, к тому же инструкции байт-кода требуют большего объема работы, чем инструкции центрального процессора. С другой стороны, в отличие от классических интерпретаторов по-прежнему имеется внутренний шаг компиляции – Python не нуждается в частом повторном анализе и разборе текста каждого оператора исходного кода. В результате чистый код Python выполняется со скоростью, находящейся между скоростью традиционного компилируемого языка и скоростью традиционного интерпретируемого языка. Компромиссы, связанные с производительностью Python, обсуждались в главе 1.

Последствия в плане разработки

Еще одно последствие модели выполнения Python заключается в том, что на самом деле нет каких-либо различий между средами разработки и выполнения. То есть системы, где компилируется и выполняется исходный код, в действительности совпадают. Такое сходство может быть чуть более важным для читателей, которые имеют опыт работы с компилируемыми языками, но в Python компилятор всегда присутствует во время выполнения и является частью системы, выполняющей программы.

В итоге цикл разработки становится гораздо более быстрым. Нет никакой необходимости в предварительной компиляции и связывании, прежде чем выполнение может начаться; просто набирайте и запускайте код. Кроме того, язык приобретает более динамичный характер – возможно и часто очень удобно, когда одни программы Python создают и запускают другие программы Python во время выполнения. Например, встроенные функции eval и exec принимают и выполняют строки, содержащие программный код Python. Именно из-за такой структуры сам Python пригоден для настройки продуктов – поскольку код Python способен изменяться на лету, пользователи могут модифицировать части Python системы на месте эксплуатации без потребности в наличии или компиляции кода всей системы.

С фундаментальной точки зрения мы на самом деле имеем в Python лишь *исполняющий механизм* – начальная стадия компиляции отсутствует и все делается во время выполнения программы. Сюда входят даже такие операции, как создание функций и классов, плюс связывание модулей. В статических языках указанные события происходят перед стадией выполнения, но в Python – во время выполнения. Как мы увидим, это поощряет более динамичный стиль программирования, нежели тот, который может быть привычным для некоторых читателей.

Разновидности модели выполнения

Теперь, когда вам известен внутренний поток выполнения, описанный в предыдущем разделе, я должен отметить, что он отражает стандартную реализацию Python в наши дни, но в действительности не является требованием самого языка Python. Из-за этого модель выполнения предрасположена к изменениям с течением времени. Фактически уже существует несколько систем, которые слегка модифицируют картину, изображенную на рис. 2.2. Прежде чем двигаться дальше, давайте исследуем самые заметные вариации из числа имеющихся.

Альтернативные реализации Python

Строго говоря, во время написания настоящего издания было доступно, по крайней мере, пять реализаций языка Python – *CPython*, *Jython*, *IronPython*, *Stackless* и *PyPy*. Несмотря на то что между ними есть много взаимно дополняющих идей и работ, каждая реализация представляет собой отдельно устанавливаемую программную систему с собственными разработчиками и пользовательской базой. В перечень других потенциальных кандидатов входят системы *Cython* и *Shed Skin*, но они обсуждаются позже как инструменты оптимизации, потому что не реализуют стандартный язык Python (первая является смесью Python/C, а вторая – неявно статически типизированной).

Выражаясь кратко, *CPython* – это стандартная реализация, которую пожелают использовать большинство читателей (если вы не уверены, то вряд ли будете исключением). Кроме того, данная версия применяется в книге, хотя представленный здесь базовый язык Python в альтернативных реализациях почти полностью сохраняется. Все другие реализации Python предназначены для специфичных целей и ролей, но они часто могут исполнять также и большинство ролей CPython. Все они реализуют тот же самый язык Python, но выполняют программы отличающимися способами.

Например, *PyPy* является заменой CPython, которая позволяет выполнять многие программы гораздо быстрее. Подобным же образом *Jython* и *IronPython* представляют собой совершенно независимые реализации Python, которые компилируют исходный код Python для разных архитектур времени выполнения, чтобы обеспечить прямой доступ к компонентам Java и .NET. Доступ к программному обеспечению Java и .NET также возможен из стандартных программ CPython – например, системы *JPyre* и *Python for .NET* позволяют стандартному коду CPython обращаться к компонентам Java и .NET. Системы *Jython* и *IronPython* предлагают более завершенные решения, предоставляя полные реализации языка Python.

Ниже представлен краткий обзор наиболее известных реализаций Python, доступных на сегодняшний день.

CPython: стандарт

Первоначальную и стандартную реализацию Python обычно называют CPython, когда ее нужно противопоставить с другими вариантами (а иначе просто Python).

Название происходит из того факта, что реализация CPython написана на переносимом языке ANSI C. Она доступна для загрузки на веб-сайте <http://www.python.org>, входит в состав дистрибутивов ActivePython и Enthought, а также автоматически установлена в большинстве систем Linux и Mac OS X. Если вы обнаружили на своем компьютере заранее установленную версию Python, то с высокой вероятностью это будет CPython при условии, что ваша компания или организация не использует Python более специализированными путями.

Если только вы не желаете снабжать приложения Java и .NET сценариями на Python или не находите преимущества Stackless или PyPy достойными внимания, тогда вероятно будете работать со стандартной системой CPython. Поскольку CPython – эталонная реализация языка, она имеет тенденцию выполняться быстрее, а также быть более полной, современной и надежной, чем альтернативные системы. Архитектура времени выполнения CPython была показана на рис. 2.2.

Jython: Python для Java

Система Jython (изначально известная как JPython) является альтернативной реализацией языка Python, направленной на интеграцию с языком программирования Java. Система Jython состоит из классов Java, которые компилируют исходный код Python в байт-код Java и затем направляют результатирующую байт-код виртуальной машине Java (Java Virtual Machine – JVM). Программисты по-прежнему размещают операторы Python в текстовых файлах .py, как обычно; система Jython по существу просто заменяет два блока справа на рис. 2.2 эквивалентами на основе Java.

Цель Jython – дать возможность снабжать сценариями Python приложения Java почти как CPython позволяет снабжать сценариями Python компоненты C и C++. Интеграция с Java удивительно бесшовная. Поскольку код Python транслируется в байт-код Java, во время выполнения он выглядит и ведет себя подобно настоящей программе Java. Сценарии Jython могут служить в качестве веб-апплетов и сервлетов, строить графические пользовательские интерфейсы на основе Java и т.д. Кроме того, Jython включает поддержку интеграции, которая разрешает коду Python импортировать и применять классы Java, как если бы они были написаны на Python, а коду Java – выполнять код Python в качестве кода на встроенным языке. Однако из-за того, что система Jython медленнее и менее надежна, чем CPython, она обычно рассматривается как инструмент, в первую очередь интересный разработчикам на Java, которые ищут язык написания сценариев, служащий интерфейсом к коду Java. Дополнительные сведения о Jython доступны на веб-сайте <http://jython.org>.

IronPython: Python для .NET

Третья реализация Python, которая новее CPython и Jython, называется IronPython и призвана снабдить программы Python возможностью интеграции с приложениями, ориентированными на работу с платформой Microsoft .NET Framework для Windows, а также с эквивалентом с открытым кодом Mono для Linux. Платформа .NET и ее исполняющая система языка программирования C# спроектированы так, чтобы быть уровнем взаимодействия между объектами, нейтральным к языкам, в духе более ранней модели COM от Microsoft. Система IronPython позволяет программам Python действовать в качестве клиентских и серверных компонентов, извлекать пользу от доступности в и из остальных языков .NET и использовать внутри кода Python технологии .NET вроде инфраструктуры *Silverlight*.

По своей реализации система IronPython очень похожа на Jython (и на самом деле была разработана тем же самым создателем) – она заменяет два блока справа на рис. 2.2

эквивалентами для выполнения в среде .NET. Подобно Jython система IronPython имеет специальную направленность – она представляет интерес главным образом для разработчиков, интегрирующих Python с компонентами .NET. Ранее разрабатываемая компанией Microsoft, а теперь являющаяся проектом с открытым кодом, система IronPython также способна получать преимущества от применения ряда важных инструментов оптимизации, обеспечивая более высокую производительность. За дополнительной информацией обращайтесь на веб-сайт <http://ironpython.net> или поищите в Интернете другие ресурсы.

Stackless: Python для обеспечения параллелизма

Остальные схемы для выполнения программ Python преследуют более специализированные цели. Например, система *Stackless Python* представляет собой усовершенствованную версию и новую реализацию стандартного языка CPython, ориентированную на предстоящий *параллелизм*. Поскольку Stackless Python не сохраняет состояние в стеке вызовов языка С, она может облегчить перенос Python в архитектуры с небольшими стеками, предлагает эффективные варианты многопроцессорной обработки и благоприятствует использованию новаторских программных структур, таких как со-программы.

Помимо всего прочего *микропотоки*, которые Stackless добавляет в Python, являются эффективной и легковесной альтернативой стандартных инструментов мозгозадачности Python, подобных потокам и процессам, а также обещают лучшую структуру программ, более читабельный код и повышенную продуктивность труда программистов. Создатель игры *EVE Online*, компания CCP Games, входит в число хорошо известных пользователей Stackless Python и в целом имеет захватывающую историю успеха в применении Python. Дополнительные сведения ищите на веб-сайте <http://stackless.com>.

PyPy: Python для обеспечения высокой скорости выполнения

Система PyPy – еще одна новая реализация стандартной системы CPython, ориентированная на *производительность*. Она предлагает быструю реализацию Python с *JIT-компилятором* (*just-in-time* – оперативный), предоставляет инструменты для модели “песочницы”, которая способна выполнять ненадежный код в защищенной среде, и по умолчанию включает поддержку для рассмотренной выше системы *Stackless Python* и ее микропотоков с целью поддержки массового параллелизма.

PyPy – последователь первоначального JIT-компилятора *Psyco*, который будет описан далее в главе, и относится к категории полных реализаций Python, построенных для обеспечения высокой скорости выполнения. В действительности JIT-компилятор является всего лишь расширением машины PVM (самый правый блок на рис. 2.2), которое транслирует порции байт-кода до самого двоичного машинного кода для более быстрого выполнения. JIT-компилятор делает это в ходе *выполнения* программы, а не на шаге компиляции, предваряющем запуск, и способен создавать машинный код, специфичный к типам, для кода на динамическом языке Python, путем отслеживания *типов данных* объектов, которые программа обрабатывает. За счет замены порций байт-кода подобным образом программа по мере своего выполнения работает все быстрее и быстрее. Вдобавок некоторые программы Python могут также занимать меньше памяти, когда выполняются под управлением PyPy.

На момент написания главы система PyPy поддерживала код Python 2.7, 3.5 и 3.6 и функционировала на платформах Intel x86 (IA-32) и x86_64 (включая Windows, Linux и недавние версии Mac) и ARM, с разрабатываемой поддержкой для PPC. Она выпол-

няет большинство кода CPython, хотя модули расширений С обычно должны быть перекомпилированы, и система PyPy имеет ряд незначительных, но тонких языковых отличий, в том числе семантику сборки мусора, которая позволяет избежать нескольких распространенных паттернов кодирования. Например, ее схема без подсчета ссылок означает, что временные файлы можно не закрывать и не сбрасывать выходные буферы немедленно, но в ряде случаев могут требоваться вызовы функций закрытия вручную.

Взамен ваш код может выполняться намного быстрее. В текущий момент заявлено, что PyPy дает ускорение в 7,6 раза по сравнению с CPython, что подтверждается целым диапазоном эталонных тестовых программ (<http://speed.pyru.org/>). В отдельных случаях способность PyPy получить преимущества от возможностей динамической оптимизации может сделать код Python таким же быстрым, как код С, а временами и быстрее. Сказанное особенно справедливо для громоздких алгоритмических и численных программ, которые иначе могли быть переписаны на С.

Например, в одном простом эталонном тесте, который мы увидим в главе 21, PyPy оказывается в 10 раз быстрее, чем CPython 2.7, и в 100 раз быстрее, чем CPython 3.X. Несмотря на то что другие эталонные тесты будут варьироваться, такое ускорение может стать неоспоримым преимуществом во многих предметных областях и возможно даже в большей степени, нежели самые передовые языковые средства. Не менее важно и то, что пространство памяти в PyPy также оптимизировано — один из опубликованных эталонных тестов потребовал 247 Мбайт и 10,3 секунды на завершение по сравнению с 684 Мбайт и 89 секунд в случае CPython.

Цепочка инструментов PyPy также достаточно общая, чтобы поддерживать дополнительные языки, включая Pyrolog (интерпретатор Prolog, написанный на Python с использованием транслятора PyPy). В текущий момент веб-сайт PyPy находится по адресу <http://pyru.org>. Обзор производительности доступен по ссылке <http://www.pyru.org/performance.html>.

Инструменты оптимизации выполнения

CPython и большинство альтернатив, рассмотренных в предыдущем разделе, реализуют язык Python похожими способами: путем компиляции исходного кода в байт-код и выполнения байт-кода на подходящей виртуальной машине. Некоторые системы, такие как гибрид Cython, транслятор Shed Skin C++ и JIT-компиляторы в PyPy и Psyco, взамен пытаются оптимизировать базовую модель выполнения. Знание этих систем на данном этапе вашей карьеры в области Python обязательным не является, но быстрый взгляд на их место в модели выполнения может помочь прояснить модель в целом.

Cython: гибрид Python/C

Система *Cython* (основанная на работе в рамках проекта *Pyxrex*) представляет собой гибридный язык, который объединяет код Python с возможностью вызова функций С и использования объявлений типов С для переменных, параметров и атрибутов классов. Код Cython может компилироваться в код С, где применяется API-интерфейс Python/C, который затем можно скомпилировать полностью. Несмотря на отсутствие полной совместимости со стандартным языком Python, система Cython может быть полезной при построении оболочек для внешних библиотек С и написании кода эффективных расширений С для Python. Текущее состояние и детали системы доступны на веб-сайте <http://cython.org>.

Shed Skin: транслятор Python в C++

Shed Skin – это развивающаяся система, в которой принят другой подход к выполнению программ Python; она пытается транслировать исходный код Python в код C++, который затем с помощью имеющегося компилятора C++ компилируется в машинный код. Как таковая, она представляет нейтральный к платформам подход к выполнению кода Python.

В настоящее время Shed Skin продолжает активно разрабатываться. Система Shed Skin поддерживает код Python 2.4–2.6 и ограничивается программами Python с неявной статической типизацией, что обычно для большинства программ, но формально не является нормальным кодом Python, поэтому мы здесь не будем вдаваться в особые детали. Тем не менее, первые результаты показывают, что потенциально система Shed Skin способна превзойти стандартный Python и Psyco-подобные расширения в терминах скорости выполнения. Текущее состояние системы можно выяснить по адресу <https://shedskin.github.io/>.

Psyco: первоначальный JIT-компилятор

Система Psyco представляет собой не очередную реализацию Python, а компонент, который расширяет модель выполнения байт-кода, чтобы программы выполнялись быстрее. На сегодняшний день Psyco является чем-то вроде *бывшего проекта*: он по-прежнему доступен для отдельной загрузки, но с развитием Python устарел и больше активно не сопровождается. Взамен его идеи были воплощены в описанной ранее более полной системе PyPy. И все же неуменьшающаяся важность идей, исследованных в проекте Psyco, делает его заслуживающим внимания.

С точки зрения рис. 2.2 система – это усовершенствование машины PVM, которое собирает и использует информацию о типах, пока программа выполняется, чтобы транслировать порции байт-кода программы до подлинного машинного кода для более быстрого выполнения. Psyco осуществляет такую трансляцию, не требуя внесения изменений в код или отдельного шага компиляции на этапе разработки.

Грубо говоря, пока ваша программа функционирует, Psyco собирает информацию о типах встречающихся объектов, которую можно применять для генерации высокоэффективного машинного кода, приспособленного под такие типы объектов. Сгенерированный машинный код заменяет соответствующую часть первоначального байт-кода, чтобы увеличить общую скорость выполнения программы. В результате посредством Psyco ваша программа со временем выполняется быстрее. В идеальных случаях под управлением Psyco некоторый код Python может стать таким же быстрым, как скомпилированный код C.

Поскольку описанная трансляция байт-кода происходит во время выполнения программы, Psyco известен как *JIT-компилятор*. Однако Psyco отличается от JIT-компиляторов для языка Java, с которыми могли сталкиваться отдельные читатели. На самом деле Psyco является *специализированным JIT-компилятором* – он генерирует машинный код, подогнанный под типы данных, которые действительно используются в программе. Скажем, если в части программы в разное время применяются отличающиеся типы данных, то Psyco может генерировать разные версии машинного кода для поддержки каждой комбинации типов.

Система Psyco была предложена для значительного повышения скорости выполнения определенного кода Python. Согласно заявлению на веб-странице системы Psyco она обеспечивает ускорение от 2 до 100 раз, обычно в 4 раза, с неизмененным интерпретатором Python и немодифицированным исходным кодом, почти как у динамически загружаемого модуля расширения C.

Не менее важно то, что наибольшее ускорение реализовано для алгоритмического кода, написанного на чистом Python — в точности той разновидности кода, которую обычно переносят на С в целях оптимизации. Дополнительные сведения о Pyso ищите в Интернете или обратитесь на веб-сайт преемника — описанного ранее проекта PyPy.

Фиксированные двоичные файлы

Временами, когда люди запрашивают “настоящий” компилятор Python, они на самом деле просто ищут способ генерации автономных двоичных исполняемых файлов из своих программ Python. Это скорее идея упаковки и поставки, чем концепция потока выполнения, но определенная связь имеется. С помощью сторонних инструментов, которые можно найти в Интернете, возможно превращение программ Python в настоящие исполняемые файлы, которые в мире Python называются *фиксированными двоичными файлами*. Результатирующие двоичные файлы могут запускаться, не требуя установки Python.

Фиксированные двоичные файлы объединяют в единственный пакет байт-код ваших файлов программ вместе с машиной PVM (интерпретатором) и любыми необходимыми файлами поддержки. Существует несколько вариаций, но конечным результатом может быть одиночная двоичная исполняемая программа (т.е. файл .exe в Windows), которую легко поставить заказчикам. На рис. 2.2 так получилось бы в случае слияния двух блоков справа (байт-кода и машины PVM) в один компонент: фиксированный двоичный файл.

На сегодняшний день генерировать фиксированные двоичные файлы способны многие системы, которые отличаются целевыми платформами и возможностями: система *py2exe* только для Windows, но с широкой поддержкой Windows; система *PyInstaller*, похожая на *py2exe*, но работает также в средах Linux и Mac OS X и может генерировать самоустанавливающиеся двоичные файлы; система *py2app* для создания приложений Mac OS X; первоначальная система *freeze*; система *cx_freeze*, которая предлагает поддержку Python 3.X и межплатформенную поддержку. Возможно, вам придется загрузить указанные инструменты отдельно от Python, но они доступны бесплатно.

Эти инструменты также постоянно развиваются, так что проверяйте их состояние на веб-сайте <http://www.python.org> или через поисковый механизм в Интернете. Чтобы вы имели представление о масштабе данных систем, необходимо отметить, что *py2exe* может фиксировать автономные программы, которые используют библиотеки для построения графических пользовательских интерфейсов *tkinter*, *PMW*, *wxPython* и *PyGTK*; программы, применяющие инструментальный набор для программирования игр *pygame*; клиентские программы *win32com* и многие другие.

Фиксированные двоичные файлы — не то же самое, что и выход настоящего компилятора; они выполняют байт-код через виртуальную машину. Следовательно, не считая возможного улучшения показателей начального запуска, фиксированные двоичные файлы выполняются с такой же скоростью, как исходные файлы. Фиксированные двоичные файлы, как правило, не имеют малых размеров (они содержат машину PVM), но по текущим стандартам не являются необычно большими. Из-за того, что Python встраивается в фиксированный двоичный файл, он не требует установки на стороне получателя, чтобы программу можно было запустить. Вдобавок поскольку код также встраивается в фиксированный двоичный файл, он более эффективно скрывается от получателей.

Такая схема упаковки в один файл особенно привлекательна для разработчиков коммерческого ПО. Скажем, программа с пользовательским интерфейсом, написан-

ным на Python и основанным на инструментальном наборе tkinter, может быть зафиксирована в виде исполняемого файла и поставлена как самостоятельная программа на компакт-диске или через веб-сеть. Для выполнения поставленной программы конечным пользователям не придется устанавливать Python (или даже знать о нем).

Будущие возможности?

Наконец, обратите внимание, что обрисованная здесь модель выполнения в действительности является артефактом текущей реализации Python, а не самого языка. Например, не исключено, что в течение срока актуальности данной книги может появиться полный традиционный компилятор для трансляции исходного кода Python в машинный код.

Также в будущем могут быть приняты новые форматы байт-кода и варианты реализации.

- Продолжающийся проект Parrot направлен на предоставление общего формата байт-кода, виртуальной машины и приемов оптимизации для разнообразных языков программирования, включая Python. Собственная машина PVM выполняет код Python более эффективно, чем Parrot, но неясно, каким образом Parrot будет развиваться конкретно в отношении Python. Детальные сведения доступны на веб-сайте <http://parrot.org>.
- В старом проекте Unladen Swallow (проекте с открытым кодом, разрабатываемом инженерами из Google) стремились ускорить стандартный Python, по крайней мере, в 5 раз, и сделать его достаточно быстрым для того, чтобы заменить язык C во многих контекстах. Это была ветвь оптимизации CPython (а именно – Python 2.6), предназначенная для обеспечения совместимости и еще большей скорости благодаря добавлению JIT-компилятора к стандартному Python. В 2011 году стало ясно, что проект закрыли (согласно отозванному предложению по улучшению Python он “повторил судьбу попугая породы ‘норвежские голубые’ из скетча ‘Мертвый попугай’ в сериале ‘Летающий цирк Монти Пайтона’”). Тем не менее, полученные уроки могут быть задействованы в других формах; поищите в веб-сети сведения о неудавшихся разработках.

Хотя будущие схемы реализации могут кое в чем изменить структуру исполняющей среды Python, вполне вероятно, что компилятор в байт-код останется стандартом в течение некоторого времени. Переносимость и гибкость байт-кода в плане выполнения являются важными характеристиками многих систем Python. Кроме того, добавление объявлений с ограничениями типов для поддержки статической компиляции, скорее всего, нарушит большую часть гибкости, лаконичности, простоты и общего духа написания кода Python. Вследствие исключительно динамической природы Python любая будущая реализация, вероятно, сохранит многие артефакты текущей машины PVM.

Резюме

В главе была представлена модель выполнения Python, т.е. показано, каким образом Python выполняет свои программы, а также исследованы распространенные вариации этой модели: JIT-компиляторы и т.д. Хотя для написания сценариев Python знать внутреннее устройство Python вовсе не обязательно, ознакомление с темами этой главы поможет по-настоящему понять, как программы выполняются после их готовности. В следующей главе вы фактически начнете запускать собственный код. Но сначала ответьте на обычные контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Что такое интерпретатор Python?
2. Что такое исходный код?
3. Что такое байт-код?
4. Что такое PVM?
5. Назовите две или большее число вариаций стандартной модели выполнения Python.
6. В чем отличия между CPython, Jython и IronPython?
7. Что такое Stackless и PyPy?

Проверьте свои знания: ответы

1. Интерпретатор Python – это программа, которая выполняет написанные вами программы Python.
2. Исходный код представляет собой операторы, которые вы пишете для своей программы – они хранятся в текстовых файлах, обычно имеющих расширение .py.
3. Байт-код – это низкоуровневая форма программы, получаемая в результате ее компиляции Python. Байт-код автоматически сохраняется в файлах с расширением .pyc.
4. PVM – это виртуальная машина Python (Python Virtual Machine), т.е. исполняющий механизм Python, который интерпретирует скомпилированный байт-код.
5. Вариациями модели выполнения являются Psyco, Shed Skin и фиксированные двоичные файлы. Вдобавок указанные в следующих двух ответах альтернативные реализации Python также в определенной степени модифицируют модель выполнения, заменяя байт-код и виртуальные машины или добавляя инструменты JIT-компиляторы.
6. CPython – стандартная реализация языка. Jython и IronPython реализуют программы Python для использования соответственно в средах Java и .NET; они являются альтернативными компиляторами для Python.
7. Stackless – это усовершенствованная версия Python, направленная на обеспечение параллелизма, а PyPy – новая реализация Python, нацеленная на обеспечение высокой скорости выполнения. PyPy также является преемником Psyco и включает в себя концепции JIT-компилятора, который впервые был разработан в рамках Psyco.

Как пользователь выполняет программы

И так, пришло время начать выполнять какой-то код. Теперь, обладая контролем над моделью выполнения, вы окончательно готовы приступить к реальному программированию на Python. В данный момент я предполагаю, что Python на вашем компьютере уже установлен; если же нет, тогда ознакомьтесь с советами по установке и конфигурированию для различных платформ, приведенными в начале предыдущей главы и в приложении А второго тома. Ваша цель здесь – научиться запускать программный код Python.

Сообщить Python о необходимости выполнения кода, который вы набираете, можно многими способами. В этой главе обсуждаются все употребительные приемы запуска программ. Попутно вы узнаете, как набирать код *интерактивно*, и каким образом сохранять его в *файлы*, чтобы запускать с желаемой частотой разнообразными путями: с помощью командных строк системы, щелчков на значках, импорта модулей, вызовов *exec*, пунктов меню в графическом пользовательском интерфейсе IDLE и т.д.

Как и с предыдущей главой, если вы имеете опыт программирования на других языках и хотите незамедлительно погрузиться в сам Python, то можете бегло просмотреть текущую главу и перейти к изучению главы 4. Но не пропускайте описание подготовительных работ и соглашений в начале главы, обзор приемов отладки или введение в импорт модулей – важные темы для понимания архитектуры программ Python, которых мы будем касаться лишь в более поздних частях книги. Я также рекомендую просмотреть разделы, посвященные IDLE и другим IDE-средам, чтобы вы знали, какие инструменты доступны, когда вы начнете разрабатывать более сложные программы Python.

Интерактивная подсказка

В данном разделе вы ознакомитесь с основами интерактивного написания кода. Поскольку это наш первый взгляд на выполнение кода, мы также раскроем здесь несколько подготовительных работ, таких как настройка рабочего каталога и пути поиска в системе, а потому обязательно прочитайте раздел, если являетесь новичком в программировании. В разделе также объясняются соглашения, используемые повсюду в книге, поэтому большинству читателей необходимо хотя бы бегло просмотреть его.

Запуск интерактивного сеанса

Возможно, самый простой способ выполнения программ Python заключается в том, чтобы набирать их в интерактивной командной строке Python, иногда называемой *интерактивной подсказкой*. Существует несколько способов запуска такой командной строки: в IDE-среде, из системной консоли и т.д. Предполагая, что в системе присутствует интерпретатор Python, установленный как исполняемая программа, наиболее нейтральный к платформам способ запуска интерактивного сеанса интерпретатора предусматривает ввод в командной строке операционной системы просто `python` без каких-либо аргументов. Например:

```
% python
Python 3.3.0 (v3.3.0:bd8af90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit
...
Type "help", "copyright", "credits" or "license" for more information.
Для получения дополнительной информации наберите help, copyright, credits
или license.
>>> ^Z
```

Ввод слова `python` в приглашении командной оболочки вашей системы начинает интерактивный сеанс Python; символом % в этой книге обозначается обобщенное приглашение на ввод команд – вы не набираете его самостоятельно. В среде Windows выход из сеанса производится нажатием клавиатурной комбинации <Ctrl+Z>; в среде Unix взамен применяется <Ctrl+D>.

Понятие *приглашения командной оболочки системы* является обобщенным, но способ получения доступа к нему доступ зависит от платформы.

- В Windows вы можете ввести `python` в консольном окне DOS – программе, которая имеет имя `cmd.exe` и обычно известна как окно командной строки. Дополнительные сведения о запуске окна командной строки приведены во врезке “Где находится окно командной строки в Windows?” далее в главе.
- В Mac OS X вы можете запустить интерактивный интерпретатор Python, дважды щелкнув на пункте Applications⇒Utilities⇒Terminal (Приложения⇒Служебные⇒Терминал) и затем введя `python` в открывшемся окне.
- В Linux (и других разновидностях Unix) вы можете ввести команду `python` в окне командной оболочки или терминала (например, в `xterm` или в консоли, выполняющей командную оболочку вроде `ksh` или `csh`).
- В других системах могут использоваться похожие или специфичные для платформы методы. Скажем, запуск интерактивного сеанса на карманном устройстве может требовать щелчка на значке Python в соответствующем окне.

На большинстве платформ запустить интерактивную подсказку можно дополнительными путями, не требующими ввода команды, но они варьируются в зависимости от платформы даже еще более широко.

- В Windows 7 и предшествующих версиях кроме ввода `python` в окне командной строки начинать похожие интерактивные сеансы можно также, запуская графический пользовательский интерфейс IDLE (обсуждается позже) или выбирай пункт Python (command line) (Python (командная строка)) для нужной версии Python, который доступен через меню Start (Пуск), как было показано на рис. 2.1. В обоих случаях откроется интерактивная подсказка Python с той же функциональностью, что и получаемая посредством команды `python`.

- В Windows 8 и последующих версиях кнопка Start может отсутствовать, но есть другие способы получения описанного выше инструмента, включая плитки, средство поиска, проводник файлов и интерфейс All apps (Все приложения) на экране Start (Пуск). Более подробные сведения ищите в приложении А (т том 2).
- На других платформах имеются аналогичные методы запуска интерактивного сеанса Python, не вводя команды, но они слишком специфичны, чтобы приводить их здесь; за подробной информацией обращайтесь в документацию по системе.

Всякий раз, когда вы видите приглашение `>>>`, вы находитесь в интерактивном сеансе интерпретатора Python — здесь можно набирать любой оператор или выражение Python и выполнять его немедленно. Вскоре мы займемся этим, но сначала нужно утрясти несколько деталей, связанных с запуском, чтобы удостовериться, что все читатели готовы.

Где находится окно командной строки в Windows?

Так каким же образом запустить интерфейс командной строки в Windows? Некоторым читателям, работающим в среде Windows, это известно, но разработчики, применяющие Unix, и начинающие могут о нем не знать; окно командной строки в Windows не настолько заметно, как окна терминала и консоли в системах Unix. Ниже приведены указания относительно того, где в Windows искать окно командной строки, что слегка отличается от версий к версии.

В Windows 7 и предшествующих версиях это окно находится обычно в разделе Accessories (Стандартные) меню Start⇒All Programs (Пуск⇒Все программы). Его также можно запустить, введя cmd в диалоговом окне, которое открывается в результате выбора пункта меню Start⇒Run... (Пуск⇒Выполнить...), или в поле поиска меню Start. При желании более быстрого доступа можно создать ярлык на рабочем столе.

В Windows 8 и последующих версиях окно командной строки можно найти в меню, открывающемся в результате щелчка правой кнопкой мыши на предварительном просмотре в нижнем левом углу экрана; в разделе Windows System (Система Windows) окна All apps, доступного по щелчку правой кнопкой мыши на экране Start; либо набрав cmd или commandprompt в поле поиска панели Charms, которая раскрывается в правом верхнем углу экрана. Вероятно, существуют дополнительные маршруты, и сенсорные экраны предлагают похожий доступ. И если вы не хотите помнить обо всем этом, тогда просто закрепите панель задач на рабочем столе для легкого доступа в следующий раз.

Описанные процедуры с течением времени могут измениться и возможно даже для каждого компьютера и пользователя. Однако я стараюсь не привязывать книгу к Windows, а потому данная тема затрагивается лишь кратко. В случае сомнений обращайтесь в справочную систему.

Примечание для пользователей Unix, читающих эту врезку и чувствующих себя явно не в своей стихии: вас может заинтересовать система *Cygwin*, которая предоставляет полную командную строку Unix для Windows. Дополнительные указания приведены в приложении А второго тома.

Пути поиска в системе

Когда мы набирали `python` в предыдущем разделе для запуска интерактивного сеанса, то полагались на тот факт, что система найдет программу Python в своем пути поиска. В зависимости от версии Python и платформы, если вы не установили переменную среды PATH своей системы так, чтобы она включала каталог, где установлен Python, то к слову `python` может потребоваться добавить полный путь к исполняемому файлу Python. В Unix, Linux и похожей среде таким каталогом часто оказывается `/usr/local/bin/python` или `/usr/bin/python3`. В среде Windows он может выглядеть как `C:\Python33\python` (для версии Python 3.3):

```
c:\code> c:\python33\python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

В качестве альтернативы вы можете с помощью команды `cd` перейти в каталог, где установлен Python, и затем ввести `python` – например, вот команда `cd c:\python33` в Windows:

```
c:\code> cd c:\python33
c:\Python33> python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

Но вполне вероятно, что в итоге вы захотите установить переменную среды PATH, чтобы достаточно было вводить просто `python`. Если вы не знаете, что собой представляет переменная PATH или как ее устанавливать, тогда обратитесь в приложение А второго тома – в нем описаны переменные среды, использование которых меняется от платформы к платформе, а также аргументы командной строки Python, применяемые в книге не особенно часто. Коротко для пользователей Windows: просмотрите дополнительные параметры системы, щелкнув на элементе Система в панели управления. Если вы используете Python 3.3 или последующей версии, то установка переменной PATH в Windows происходит автоматически, как объясняется далее.

Новые возможности для Windows в версии Python 3.3: переменная среды PATH и запускающий модуль

В предшествующем разделе и большей части текущей главы в целом описано общее положение дел для всех версий Python 2.X и 3.X до версии Python 3.3. Начиная с Python 3.3, установщик Windows предлагает возможность *автоматического* добавления каталога с Python 3.3 в переменную среды PATH, когда это включено в окнах установщика. Если вы задействуете такую возможность, то для выполнения команды `python` набирать путь к каталогу или выдавать команду `cd`, как в предыдущем разделе, не придется. Удовствуйтесь в том, что во время установки выбрали эту возможность, если она нужна, т.к. по умолчанию она отключена.

Более существенно то, что Python 3.3 для Windows поставляется с новым автоматически устанавливаемым *запускающим модулем Windows* – системой, поступающей с новыми исполняемыми программами, ру с консолью и ру без, которые размещаются в каталогах внутри пути поиска системы и потому могут запускаться сразу же без какого-либо конфигурирования переменной среды PATH, команды `cd` или префиксов в виде пути к каталогу:

```
c:\code> py
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z

c:\code> py -2
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z

c:\code> py -3.1
Python 3.1.4 (default, Jun 12 2011, 14:16:16) [MSC v.1500 64 bit (AMD64)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

В двух последних командах видно, что эти исполняемые программы также принимают в командной строке номера версий Python (и в строках `#!` стиля Unix в начале сценариев, как обсуждается позже). В добавок с их помощью открываются файлы Python при двойном щелчке на них подобно первоначальной исполняемой программе `python`, которая по-прежнему доступна и работает, но отчасти вытеснена новыми программами запускающего модуля.

Запускающий модуль – стандартная часть Python 3.3, и он доступен как автономная установка для применения с другими версиями. Мы еще рассмотрим новый запускающий модуль в этой и будущих главах, в том числе его поддержку строки `#!`. Тем не менее, поскольку он интересует только пользователей Windows и даже в данной группе присутствует только в версии Python 3.3 или там, где он был установлен отдельно, почти все детали о запускающем модуле собраны в приложении Б второго тома.

Если вы работаете под управлением Windows с Python 3.3 или последующей версией, я предлагаю заглянуть в приложение Б во втором томе, т.к. оно предлагает альтернативный и в некоторых отношениях лучший способ выполнения командных строк и сценариев Python. На базовом уровне пользователи запускающего модуля в большинстве приведенных в книге системных команд могут вводить `py` вместо `python` и избежать ряда шагов конфигурирования. На компьютерах с множеством установленных версий Python запускающий модуль обеспечивает более явный контроль над тем, какая версия Python выполняет имеющийся код.

Где выполнять: каталоги для кода

Теперь, когда мы начали рассматривать, *как* выполнять код, следует сказать несколько слов о том, *где* его выполнять. Ради упрощения в главе и во всей книге код будет выполняться в рабочем каталоге (он же *папка*) по имени `C:\code` на компьютере Windows – подкаталоге в корне главного диска. Именно здесь будет запускаться большинство интерактивных сеансов, а также сохраняться и выполняться большинство файлов сценариев. Кроме того, в этот каталог будут помещаться файлы, создаваемые примерами кода.

Ниже приведено несколько указаний, которые помогут настроить рабочий каталог на компьютере.

- В системе Windows вы можете создать рабочий каталог для кода в проводнике файлов или в окне командной строки. В проводнике файлов ищите кнопку Новая папка, просмотрите меню Файл или попробуйте щелкнуть правой кнопкой мыши. В окне командной строки выполните команду `mkdir` после команды `cd` для перехода в желаемый корневой каталог (например, `cd c:\` и `mkdir code`).

Вы можете поместить свой рабочий каталог куда угодно и назначить ему любое имя; он не обязан быть C:\code. Использование одного каталога поможет отслеживать свои работы и упрощает некоторые задачи. Дополнительные советы для системы Windows ищите во врезке “Где находится окно командной строки в Windows?” ранее в главе и в приложении А второго тома.

- В системе, основанной на Unix (включая Mac OS X и Linux), рабочий каталог можно расположить в /usr/home и создать с помощью команды mkdir в окне командной оболочки или с помощью проводника файлов с графическим пользовательским интерфейсом, специфичного для имеющейся платформы. Unix-подобная система Cygwin для Windows также похожа, хотя имена каталогов могут быть другими (скажем, /home и /cygdrive/c).

Вы также можете хранить свой код в каталоге, где установлен Python (например, C:\Python33 в Windows), чтобы упростить некоторые командные строки до корректировки переменной среды PATH, но вероятно не должны так поступать – он предназначен для самого Python, и ваши файлы могут потеряться из-за перемещения или удаления Python.

После того, как рабочий каталог создан, всегда начинайте с перехода в него при проработке примеров, приводимых в книге. Приглашения к вводу, в которых присутствует имя каталога, где выполняется код, соответствуют моему рабочему каталогу на компьютере Windows; когда вы видите C:\code> или %, мысленно подставляйте имя и расположение своего рабочего каталога.

Что не набирать: приглашения к вводу и комментарии

Что касается приглашений к вводу, в книге иногда встречается обобщенное приглашение %, а временами указывается полный каталог C:\code> в форме Windows. Первый вариант означает независимость от платформы (и обусловлен предшествующими изданиями книги, в которых применялась система Linux), а второй используется в контекстах, специфичных для Windows. Ради читабельности я также добавляю пробел после приглашений к вводу. Символ % в начале системной команды обозначает приглашение к вводу системы, каким бы оно ни было на вашем компьютере. Например, на моем компьютере % означает C:\code> в окне командной строки и \$ в установленной системе Cygwin.

Для начинающих: не набирайте символ % (или приглашение к вводу системы C:\code, когда оно указано), присутствующий в листингах взаимодействия с интерпретатором – это текст, выводимый системой. Набирайте только текст, находящийся *после* приглашений к вводу системы. Подобным же образом не набирайте символы >>> и . . . в начале строк в листингах взаимодействия с интерпретатором – они представляют собой приглашения, которые Python автоматически отображает в качестве визуальных подсказок для интерактивного ввода кода. Набирайте только текст, указанный *после* таких приглашений Python. Скажем, приглашение . . . применяется для строк продолжения в некоторых командных оболочках, но не в IDLE, и оно показано далеко не во всех листингах в книге; не набирайте его, если оно отсутствует в вашем интерфейсе.

Чтобы помочь вам запомнить все сказанное выше, пользовательский ввод в книге представлен шрифтом с полужирным начертанием, а приглашения к вводу – шрифтом с обычным начертанием. В ряде систем приглашения к вводу могут отличаться (например, в ориентированной на производительность реализации PyPy, описанной в

главе 2, используется четырехсимвольные приглашения >>> и), но применимы те же самые правила. Также помните о том, что команды, набранные после приглашений к вводу системы и Python, предназначены для немедленного выполнения и, как правило, не сохраняются в файлах исходного кода, которые мы будем создавать; вскоре мы увидим, почему такое различие имеет значение.

Продолжая в том же духе, вы обычно не должны набирать текст, который начинается с символа # в листингах — как вы узнаете, такой текст является *комментариями*, а не исполняемым кодом. За исключением случаев, когда символ # используется для ввода директивы в начале сценария для Unix или запускающего модуля Python 3.3 для Windows, вы можете благополучно игнорировать текст, который следует за ним (дополнительные сведения будут даны позже в главе и в приложении Б второго тома).



Важно хотя бы на начальной стадии работы с примерами набирать код вручную, чтобы получить представление о синтаксисе и возможных ошибках. Некоторые примеры будут приводиться либо сами по себе, либо в имеющихся файлах, доступных в пакете примеров для книги, и мы часто будем переключаться между форматами листингов; при наличии сомнений, если вы видите >>>, то это значит, что код должен набираться вручную.

Интерактивное выполнение кода

Теперь, когда предварительные условия удовлетворены, давайте перейдем к набору фактического кода. Сразу после запуска интерактивный сеанс Python выводит две строки с информацией о номере версии Python и несколькими подсказками, показанными ранее (которые в большинстве примеров опущены ради экономии места), затем с помощью >>> приглашает к вводу и ожидает от вас набора нового оператора или выражения Python.

При интерактивной работе результаты выполнения кода отображаются ниже входных строк >>> после нажатия клавиши <Enter>. Например, далее представлены результаты выполнения двух Python-операторов print (на самом деле print является вызовом функции в версии Python 3.X, но не в Python 2.X, поэтому круглые скобки обязательны только в Python 3.X):

```
% python
>>> print('Hello world!')
Hello world!
>>> print(2 ** 8)
256
```

Вот и все — мы только что выполнили код Python (а вы ожидали *испанскую инквизицию*?). Пока что не переживайте по поводу деталей операторов print; ив начнем углубляться в синтаксис, начиная со следующей главы. Если кратко, то они выводят строку Python и целое число, как видно в строках после каждой входной строки >>> (2 ** 8 в Python означает возвведение 2 в степень 8).

При таком интерактивном написании кода вы можете набирать столько команд Python, сколько хотите; каждая выполняется немедленно после ввода. Кроме того, поскольку интерактивный сеанс автоматически выводит результаты набираемых выражений, обычно нет необходимости явно указывать print:

```
>>> lumberjack = 'okay'
>>> lumberjack
'okay'
```

```
>>> 2 ** 8
256
>>> ^Z  # Использовать для выхода <Ctrl+D> (в Unix) или <Ctrl+Z> (в Windows)
%
```

Здесь первая строка сохраняет значение, присваивая его *переменной* (*lumberjack*), которая создается этим присваиванием; две другие набранные строки представляют собой *выражения* (*lumberjack* и 2^{**8}), результаты которых отображаются автоматически. Для выхода из интерактивного сеанса подобного рода и возвращения в окно командной оболочки системы нажмите комбинацию <Ctrl+D> на компьютере с Unix или <Ctrl+Z> на компьютере с Windows. В обсуждаемом позже графическом пользовательском интерфейсе IDLE либо нажмите комбинацию <Ctrl+D>, либо просто закройте окно.

Обратите внимание в листинге на выделенное курсивом примечание (начинающееся с символа *#*). Они будут применяться повсеместно для добавления пометок о том, что иллюстрируется, но вы не обязаны набирать их текст. Подобно подсказкам к вводу системы и Python вы не должны набирать их; текст, начинающийся с *#*, воспринимается Python как комментарий.

Код в этом сеансе мало что делает — мы просто набрали ряд операторов *print* и операторов присваивания Python вместе с несколькими выражениями, которые будут подробно исследоваться позже. Главный аспект здесь в том, что интерпретатор выполняет код, введенный в каждой строке, сразу же после нажатия клавиши <Enter>.

Скажем, когда мы набрали первый оператор *print* в приглашении >>>, вывод (строка Python) был возвращен тотчас же. Не было нужды создавать файл исходного кода и предварительно прогонять код через компилятор и компоновщик, как пришлось бы делать в случае использования языка вроде C или C++. В последующих главах вы увидите, что в интерактивной подсказке можно выполнять многострочные операторы; такие операторы выполняются немедленно после ввода всех строк и нажатия <Enter> два раза для добавления пустой строки.

Для чего нужна интерактивная подсказка?

Интерактивная подсказка сразу же выполняет код и выводит результаты, но не сохраняет код в файл. Хотя это означает, что вы не будете набирать много кода в интерактивных сеансах, интерактивная подсказка оказывается великолепным местом для *экспериментирования* с языком и *тестирования* файлов с программами на лету.

Экспериментирование

Поскольку код выполняется немедленно, интерактивная подсказка является отличным местом для проведения экспериментов с языком и будет часто применяться в книге для демонстрации небольших примеров. По сути, вот первое эмпирическое правило для запоминания: если вы сомневаетесь в понимании работы той или иной порции кода Python, тогда запустите интерактивную подсказку, выполните код и посмотрите, что произойдет.

Например, пусть вы читаете код программы Python и добираетесь до выражения наподобие *'Spam!' * 8*, смысл которого вам не понятен. С одной стороны, вы можете потратить десятки минут, перелопачивая руководства, книги и веб-сеть в попытке выяснить, что делает код. С другой стороны, вы можете просто выполнить указанное выражение интерактивным образом:

```
% python
>>> 'Spam!' * 8          # Изучение методом проб
'Spam!Spam!Spam!Spam!Spam!Spam!Spam!'
```

Незамедлительная обратная связь, которую вы получаете в интерактивной подсказке, часто является самым быстрым способом сделать вывод о том, что делает порция кода. Здесь совершенно ясно, что код осуществляет повторение строки: в Python символ * означает умножение для чисел, но повторение для строк – оно похоже на многократное сцепление строки самой с собой (более подробно о строках пойдет речь в главе 4).

Экспериментируя подобным образом, вряд ли вы что-то выведете из строя, во всяком случае, пока еще. Чтобы нанести реальный вред вроде удаления файлов и выполнения команд оболочки вы должны явно импортировать модули (вам многое предстоит узнать о системных интерфейсах Python, прежде чем вы начнете представлять такую опасность!). Выполнять прямолинейный код Python почти всегда безопасно.

Например, посмотрите, что произойдет, когда вы *совершаете ошибку* в интерактивной подсказке:

```
>>> x                                # Совершена ошибка
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'X' is not defined
Трассировка (самый последний вызов указан последним) :
  Файл <stdin>, строка 1, в <модуль>
Ошибка в имени: имя X не определено
```

Использование переменной до того, как ей было присвоено значение, в Python всегда приводит к ошибке – в противном случае, если бы переменные заполнялись своими стандартными значениями, то некоторые ошибки не удалось бы выявить. Таким образом, вы обязаны инициализировать счетчики нулями, прежде чем сможете увеличивать их, инициализировать списки до их расширения и т.д.; вы не объявляете переменные, но им должны быть присвоены значения до того, как их можно будет извлекать.

Дополнительные сведения об этом будут даны позже, а сейчас важно понимать, что такая ошибка не приводит к аварийному отказу Python или вашего компьютера. Взамен вы получаете содержательное сообщение, указывающее на ошибку и строку кода, где она была допущена, а вы можете продолжить работу в сеансе или сценарии. На самом деле после того, как вы освоитесь с Python, его сообщения об ошибках зачастую способны снабдить всей необходимой поддержкой отладки (возможности отладки более подробно описаны во врезке “Отладка кода Python” далее в главе).

Тестирование

Помимо того, что интерактивный интерпретатор служит инструментом для экспериментирования, пока вы изучаете язык, он также является идеальным местом для тестирования кода, написанного в файлах. Вы можете импортировать свои файлы модулей интерактивным образом и прогнать тесты на определяемых ими инструментах, набирая вызовы в интерактивной подсказке на лету.

Скажем, следующие команды тестируют функцию в готовом модуле, который поставляется вместе с Python в его стандартной библиотеке (она выводит имя каталога, где вы работаете в текущий момент, с удвоенной обратной косой чертой вместо одной), но вы можете поступать аналогично, когда начнете создавать собственные файлы модулей:

```
>>> import os
>>> os.getcwd()                      # Тестирование на лету
'c:\\\\code'
```

В более общем смысле интерактивная подсказка представляет собой место для тестирования программных компонентов независимо от их источника – вы можете импортировать и тестируйте функции и классы в своих файлах Python, набирать вызовы связанных функций C, упражняться с классами Java под Jython и т.д. Частично из-за своей интерактивной природы Python поддерживает экспериментальный и исследовательский стиль программирования, который вы считете удобным в начале работы. Несмотря на то что программисты на Python также тестируют с помощью кода внутри файлов (позже в книге вы узнаете способы его упрощения), для многих интерактивная подсказка по-прежнему остается первой ступенью защиты в плане тестирования.

Замечания по использованию: интерактивная подсказка

Хотя интерактивная подсказка проста в применении, есть несколько полезных советов, которые должны иметь в виду начинающие. Для справки ниже описаны распространенные заблуждения, знание которых избавит вас от возможных проблем.

- Набирайте только команды Python. В первую очередь запомните, что в приглашении `>>>` можно набирать только код Python, а не системные команды. Существуют способы запуска системных команд из кода Python (например, с помощью `os.system()`), но они не сводятся к простому набору самих команд.
- Операторы `print` требуются только в файлах. Поскольку интерактивный интерпретатор автоматически выводит результаты выражений, нет необходимости набирать полные операторы `print`. Эта особенность удобна, но может сбивать с толку пользователей, когда они переходят к написанию кода в файлах: внутри файла кода для просмотра вывода вы обязаны использовать операторы `print`, потому что результаты выражений автоматически не отображаются. Запомните, что вы должны применять операторы `print` в файлах, но в интерактивной подсказке они не обязательны.
- Не делайте отступы в интерактивной подсказке (пока что). При наборе программ Python, либо интерактивно, либо в текстовом файле, обязательно начинайте невложенные операторы с колонки 1 (т.е. слева). В противном случае Python может вывести сообщение о синтаксической ошибке (`SyntaxError`), т.к. пробелы слева от кода считаются отступом, который группирует вложенные операторы. Вплоть до главы 10 все набираемые вами операторы будут невложеными, так что на данный момент правило охватывает весь код. Запомните, что ведущие пробелы приводят к генерации сообщения об ошибке, поэтому не начинайте набор в интерактивной подсказке с пробела или табуляции, если только не имеете дела с вложенным кодом.
- Следите за изменениями приглашения к вводу для составных операторов. Составные (многострочные) операторы начнут встречаться в главе 4 и основательно использоваться в главе 10, но сейчас вы должны знать, что при интерактивном наборе второй и последующих строк составного оператора приглашение к вводу может измениться. В простом интерфейсе окна командной оболочки при вводе второй и последующих строк интерактивная подсказка изменяется с `>>>` на `. . . ;`, в графическом пользовательском интерфейсе IDLE в строках после первой автоматически делаются отступы.

В главе 10 вы увидите, почему это имеет значение. А пока если при вводе своего кода вы сталкиваетесь с приглашением `. . .` или пустой строкой, то вероятно каким-то образом запутали интерактивный сеанс Python, заставив его считать,

что набирается многострочный оператор. Попробуйте нажать клавишу <Enter> или комбинацию <Ctrl+C>, чтобы вернуться к главному приглашению. Строки приглашений >>> и ... также можно изменить (они доступны во встроенным модуле `sys`), но в листингах примеров предполагается, что они не изменились.

- Завершайте составные операторы в интерактивной подсказке пустой строкой. Вставка пустой строки (нажатием клавиши <Enter> в начале строки) в интерактивной подсказке необходима для сообщения интерактивному сеансу Python о том, что вы закончили набирать многострочный оператор. Таким образом, чтобы запустить многострочный оператор, вы должны нажать клавишу <Enter> два раза. Напротив, наличие пустых строк в файлах не обязательно и они просто игнорируются. Если вы не нажмете клавишу <Enter> два раза в конце составного оператора при интерактивной работе, то застрянете в подвешенном состоянии, т.к. интерактивный интерпретатор ничего не будет делать, а ждать, когда вы снова нажмете клавишу <Enter>!
- Интерактивная подсказка выполняет по одному оператору за раз. В интерактивной подсказке вы обязаны выполнить один оператор до полного завершения и только затем набирать другой. Для простых операторов это вполне естественно, потому что нажатие клавиши <Enter> приводит к выполнению введенного оператора. Однако в случае составных операторов не забывайте, что вы должны ввести пустую строку для завершения оператора и его запуска, прежде чем сможете приступить к набору следующего оператора.

Ввод множества операторов

Я получил много сообщений электронной почты от читателей, которые обожглись на последних двух пунктах, поэтому они похоже заслуживают особого внимания, хотя я рисую чересчур повторяться. Многострочные (они же составные) операторы будут представлены в следующей главе, а позже в книге их синтаксис анализируется более формально. Тем не менее, поскольку поведение составных операторов несколько отличается в файлах и в интерактивной подсказке, здесь есть два предостережения.

Прежде всего, удостоверьтесь в том, что в интерактивной подсказке заканчиваете многострочные составные операторы вроде циклов `for` и проверок `if` пустой строкой. Другими словами, *вы обязаны нажать клавишу <Enter> два раза*, чтобы завершить полный многострочный оператор и затем запустить его. Вот пример:

```
>>> for x in 'spam':  
...     print(x)      # Чтобы цикл выполнился, нажмите здесь <Enter> два раза  
...
```

Однако в файле сценария пустые строки после составных операторов не нужны; они требуются *только* в интерактивной подсказке. В файле пустые строки не обязательны и игнорируются, если присутствуют; в интерактивной подсказке они заканчивают многострочные операторы. Напоминание: приглашение к продолжению ... в предыдущем листинге выводится Python автоматически в качестве визуального напоминания; в вашем интерфейсе (скажем, IDLE) оно может не отображаться и временами опускается в книге, но не набирайте его в случае отсутствия.

Также не забывайте, что интерактивная подсказка выполняет только *по одному оператору за раз*: чтобы выполнить цикл или другой многострочный оператор, вы должны нажать клавишу <Enter> два раза и только затем можете набирать следующий оператор:

```
>>> for x in 'spam':  
...     print(x)      # Перед набором нового оператора нажмите <Enter> два раза  
... print('done')  
File "<stdin>", line 3  
    print('done')  
^  
SyntaxError: invalid syntax  
Синтаксическая ошибка: недопустимый синтаксис
```

Это означает, что в интерактивной подсказке вы не можете вырезать и вставить множество строк кода, если только код не содержит пустые строки после каждого составного оператора. Такой код лучше выполнять в *файле*, что и будет темой следующего раздела.

Командная строка системы и файлы

Хотя интерактивная подсказка великолепно подходит для экспериментирования и тестирования, она обладает одним крупным недостатком: набираемые в ней программы исчезают, как только интерпретатор Python их выполнит. Из-за того, что вводимый интерактивно код никогда не сохраняется в файле, вы не можете запустить его снова, не набирая повторно с нуля. Вырезание и вставка, а также повторный вызов команд могут помочь в какой-то степени, но не сильно, особенно при написании крупных программ. Чтобы вырезать и вставлять код в интерактивном сеансе, вам придется бы редактировать приглашения Python, вывод программы и т.п. – не самая современная методология разработки ПО!

Для сохранения программ на постоянной основе вам необходимо записывать код в файлы, которые обычно известны как *модули*. Модули – это просто текстовые файлы, содержащие операторы Python. Выполнение операторов в таком файле можно запрашивать у интерпретатора Python любое количество раз и разнообразными способами – в командной строке системы, посредством щелчка на значке, путем выбора пункта меню в пользовательском интерфейсе IDLE и т.д. Независимо от способа запуска Python каждый раз выполняет весь код в файле модуля от начала до конца.

Терминология в этой области может кое в чем отличаться. Например, на файлы модулей часто ссылаются как на *программы* в Python – т.е. программой считается последовательность готовых операторов, сохраненных в файле для многократного выполнения. Файлы модулей, запускаемые напрямую, иногда также называются *сценариями* – неформальный термин, который обычно обозначает файл программы верхнего уровня. Некоторые приберегают термин “модуль” для файла, импортируемого из другого файла, а “сценарий” – для главного файла программы; как правило, мы будем поступать аналогично (далее в главе вам предстоит еще узнать о том, что означают “верхний уровень”, импортование и главные файлы).

Независимо от названия в последующих разделах исследуются способы запуска кода, набранного в файлах модулей. В этом разделе вы узнаете, как запускать файлы наиболее базовым методом: перечисляя их имена в команде `python`, вводимой в командной строке системы. Хотя кому-то такой прием может показаться примитивным (и его часто удается избежать за счет применения графического пользовательского интерфейса, подобного обсуждаемому позже IDLE), для многих программистов окно командной строки системы вместе с окном текстового редактора представляют собой большую часть интегрированной среды разработки, в которой они нуждаются, и обеспечивают более прямой контроль над программами.

Первый сценарий

Итак, начнем. Откройте предпочтаемый текстовый редактор (например, *vi*, “Блокнот” или редактор IDLE), наберите следующие операторы в новом текстовом файле по имени `script1.py` и сохраните его в рабочем каталоге для кода, который вы настроили ранее:

```
# Первый сценарий Python

import sys          # Загрузить библиотечный модуль
print(sys.platform) # Возвести 2 в степень
x = 'Spam!'
print(x * 8)        # Повторить строку
```

Файл `script1.py` является нашим первым официальным сценарием Python (не считая двухстрочный в главе 2). Вы не должны особо беспокоиться о коде в этом файле, но в качестве краткого описания данный файл:

- импортирует модуль Python (библиотеки дополнительных инструментов), чтобы извлечь название платформы;
- три раза вызывает функцию `print` для отображения результатов сценария;
- использует переменную по имени `x`, созданную при ее присваивании, чтобы хранить строковый объект;
- применяет к объектам различные операции, которые мы начнем изучать в следующей главе.

Здесь `sys.platform` – просто строка, идентифицирующая вид компьютера, на котором вы работаете; она находится в стандартном модуле Python по имени `sys`, который должен быть импортирован, чтобы загрузиться (импортование подробно рассматривается позже).

Ради разнообразия в код добавлено несколько формальных *комментариев Python* – текст после символов `#`. О них упоминалось ранее, но теперь, когда они появляются в сценариях, я обязан быть более формальным. Комментарии могут встречаться в строках сами по себе или помещаться справа от кода в строке. Текст после символа `#` просто пропускается как примечание, предназначенное для чтения человеком, и не считается частью синтаксиса оператора. При копировании кода вы можете игнорировать комментарии; они имеют лишь информативный характер. Как правило, в книге для комментариев используется другой стиль форматирования, чтобы сделать их визуально более заметными, но в коде они будут выглядеть как обычный текст.

Не стоит пока концентрироваться на синтаксисе кода; весь он будет объяснен позже. Главное, что следует здесь отметить – код набирается в файле, а не в интерактивной подсказке. В процессе работы вы написали полнофункциональный сценарий Python.

Обратите внимание, что файлу модуля назначено имя `script1.py`. Как и любой файл верхнего уровня, его можно было бы назвать просто `script`, но имена файлов кода, которые вы хотите импортировать на стороне клиента, должны заканчиваться суффиксом `.py`. Импортование будет обсуждаться далее в главе. Неплохо снабжать суффиксом `.py` большинство своих файлов с кодом Python, поскольку в будущем может возникнуть желание их импортировать. Кроме того, некоторые текстовые редакторы выявляют файлы Python по наличию суффикса `.py` в их именах; если этот суффикс отсутствует, то вы можете не получить возможности вроде цветовой раскраски синтаксиса и автоматического добавления отступов.

Запуск файлов в командной строке

После сохранения текстового файла вы можете предложить Python запустить его, указав полное имя файла в первом аргументе команды `python`, как в показанной ниже команде, которая набрана в окне командной оболочки системы (не вводите ее в интерактивной подсказке Python и прочитайте следующий абзац, если команда сразу же не заработала):

```
% python script1.py  
freebsd13  
1267650600228229401496703205376  
Spam! Spam! Spam! Spam! Spam! Spam!
```

Вы можете набирать такую команду оболочки в любом месте, предоставляемой системой для ввода строк команд: окно командной строки Windows, окно `xterm` и т.п. Но удостоверьтесь в том, что запускаете ее в том же самом рабочем каталоге, где был сохранен файл сценария (может потребоваться сначала применить команду `cd`), и в приглашении системы, а не в приглашении `>>>`, отображаемом Python. Также не забудьте дополнить слово `python` в команде полным путем к каталогу, куда устанавливался Python, если переменная среды `PATH` не была сконфигурирована. Для запускающего модуля `py` в Windows поступать так не обязательно и это может не требоваться в Python 3.3 и последующих версиях.

Еще одно замечание для начинающих: не помещайте любой приведенный выше текст в файл исходного кода `script1.py`, созданный в предыдущем разделе. Данный текст является командой системы и выводом программы, а не программным кодом. Первая строка представляет собой команду оболочки, используемую для запуска файла исходного кода, а следующие за ней строки – результаты, выводимые операторами `print` в файле исходного кода. Также помните о том, что символ `%` обозначает приглашение к вводу системы – не набирайте его (не воспринимайте как ворчание, но на первых порах это удивительно часто допускаемая ошибка).

Если все работает по плану, то введенная команда оболочки заставит Python выполнить код в файле строка за строкой, и вы увидите вывод трех операторов `print` сценария – название платформы, как оно известно Python, результат возведения 2 в степень 100 и результат выражения, повторяющего строку, которое встречалось ранее (в главе 4 последние две операции рассматриваются более подробно).

Если все работает не так, как планировалось, тогда вы получите сообщение об ошибке – проверьте, что вы ввели код в точности, как было показано, и повторите попытку. В следующем разделе будут предложены дополнительные варианты и указания относительно данного процесса, а во врезке “Отладка кода Python” далее в главе речь пойдет о возможностях отладки, но в настоящий момент лучше всего механически скопировать сценарий. И если ничего не помогает, то можете также попробовать выполнить сценарий в обсуждаемом ниже графическом пользовательском интерфейсе IDLE – инструменте, который скрывает ряд деталей запуска, правда временами за счет утраты более явного контроля, имеющегося в случае применения командной строки.

Если копирование становится слишком утомительным или подверженным ошибкам, то можете загрузить готовые примеры кода, хотя на первых порах набор кода вручную помогает избегать синтаксических ошибок.

Варианты использования командной строки

Из-за того, что в данной схеме для запуска программ Python задействована командная оболочка, применим весь обычный синтаксис оболочки. Например, вы можете перенаправить вывод сценария Python в файл для использования либо инспектирования в будущем, применяя специальный синтаксис оболочки:

```
% python script1.py > saveit.txt
```

В итоге три строки вывода, показанные в предыдущем запуске, вместо отображения сохраняются в файле `saveit.txt`. Прием общеизвестен как перенаправление потока и работает для ввода и вывода в системах Windows и Unix. Он удобен при тестировании, т.к. вы можете писать программы, которые отслеживают изменения в выводе других программ. Тем не менее, перенаправление потока не имеет ничего общего с интерпретатором Python (он просто его поддерживает), а потому мы не будем здесь вдаваться в особые детали.

На платформе *Windows* пример работает аналогично, но приглашение к вводу системы обычно будет другим, как объяснялось ранее:

```
C:\code> python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam!
```

Если вы не настраивали переменную среды `PATH`, чтобы включить полный путь к каталогу с `python`, тогда укажите путь в команде или перейдите сначала в каталог, где находится `python`:

```
C:\code> C:\python33\python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam!
```

В качестве альтернативы, если вы используете запускающий модуль *Windows*, появившийся в Python 3.3, то команда `py` дает тот же самый результат, но не требует включения каталога или настройки переменной среды `PATH` и позволяет указывать номер версии Python:

```
c:\code> py -3 script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam!
```

В новых версиях *Windows* вы также можете ввести только имя *сценария*, опустив имя интерпретатора Python. Поскольку в новейших системах Windows применяются ассоциации расширений файлов для поиска программы, с помощью которой запускается файл, для запуска файла `.py` явно указывать `python` или `py` не нужно. Скажем, на большинстве компьютеров с *Windows* предыдущую команду можно упростить, как показано ниже, и файл `script1.py` автоматически запустится под управлением `python` до версии Python 3.3 и `py` для Python 3.3 и последующих версий – как если бы вы дважды щелкнули на значке файла в проводнике:

```
C:\code> script1.py
```

Наконец, не забывайте указывать полный путь к файлу сценария, если вы находитесь не в рабочем каталоге. Например, следующая команда, запущенная в каталоге `D:\other`, выполняет файл `script1.py`, расположенный в другом месте, но предполагает, что была настроена переменная среды `PATH`:

```
C:\code> cd D:\other  
D:\other> python c:\code\script1.py
```

Если же переменная среды PATH не включает каталог, где установлен Python, вы не используете запускающий модуль Windows (.py) и текущим является ни каталог с Python, ни каталог с файлом сценария, то понадобится указать полные пути для `python` и для `script1.py`:

```
D:\other> C:\Python33\python c:\code\script1.py
```

Замечания по использованию: командная строка и файлы

Запуск файлов программ в командной строке системы – довольно простой вариант, особенно если ранее вам приходилось работать с командной строкой. Пожалуй, это также самый переносимый способ запуска программ Python, т.к. практически на каждом компьютере присутствует некоторое представление о командной строке и структуре каталогов. Однако ниже описаны распространенные ловушки, в которые могут попасть новички, и советы, как их избежать.

- Будьте осторожны с автоматически назначаемыми расширениями в Windows и IDLE. Если для набора кода в файлах программ на компьютере Windows вы применяете “Блокнот”, тогда при сохранении файла удостоверьтесь в том, что выбран вариант Все файлы (*.*), и явно назначайте для имени файла суффикс .py. В противном случае “Блокнот” сохранит файл с расширением .txt (т.е. как `script1.py.txt`), затрудняя работу с ним в ряде схем; скажем, файл не будет импортируемым.

Хуже того, по умолчанию Windows скрывает расширения файлов, так что если вы не изменили параметры отображения, то можете даже не заметить, что поместили код в текстовый файл, а не в файл Python. Выдать это может значок файла – если на нем нет изображения какой-то разновидности змеи, то могут возникнуть затруднения. Другими симптомами проблемы являются отсутствие цветовой раскраски синтаксиса в IDLE и открытие файла для редактирования вместо его запуска при двойном щелчке на его имени.

Подобным же образом Microsoft Word по умолчанию добавляет расширение .doc, и что хуже – помещает в файл символы форматирования, которые с точки зрения синтаксиса Python незаконны. Запомните эмпирическое правило: при сохранении файла в Windows всегда выбирайте вариант Все файлы (*.*) либо используйте более дружественный к программистам текстовый редактор, такой как IDLE. Среда IDLE даже не добавляет суффикс .py автоматически – возможность, которая одним программистам нравится, но другим нет.

- Применяйте расширения файлов и пути к каталогам в командной строке системы, но не при импортировании. Не забывайте набирать полное имя файла в командной строке системы, т.е. используйте `python script1.py`, а не `python script1`. И наоборот, не указывайте суффиксы .py и пути к каталогам в Python-операторах `import`, которые встречаются позже в главе (например, `import script1`). Это может казаться очевидным, но путаница между двумя ситуациями – распространенная ошибка.

Наличие системного приглашения к вводу означает, что вы находитесь в командной оболочке системы, а не в Python, и потому правила поиска файлов модулей Python неприменимы. По этой причине вы обязаны включать расширение .py и при необходимости полный путь к файлу, который хотите запустить. Скажем, для запус-

ка файла, который хранится в каталоге, отличающемся от того, где вы работаете, вы должны указать полный путь к нему (например, `python d:\tests\spam.py`). Тем не менее, внутри кода Python вы можете просто использовать `import spam` и полагаться на то, что Python отыщет ваш файл модуля.

- Применяйте операторы `print` в файлах. Да, об этом уже упоминалось, но с ним связана настолько часто допускаемая ошибка, что уместно повториться хотя бы раз. В отличие от набора кода в интерактивной подсказке, для организации вывода в файлах программ в общем случае вы должны использовать операторы `print`. Если вы не видите вывода, тогда удостоверьтесь в наличии операторов `print` в файле. В интерактивном сеансе Python операторы `print` не являются обязательными, т.к. результаты выражений выводятся автоматически; никакого вреда они не наносят, а лишь требуют излишнего набора.

Исполняемые сценарии в стиле Unix: # !

Следующий прием запуска в действительности представляет собой специализированную форму предыдущего приема и, несмотря на название раздела, в настоящее время может применяться для запуска файлов программ под управлением Unix и Windows. Поскольку он происходит из Unix, давайте с этого и начнем.

Основы сценариев Unix

Если вы собираетесь использовать Python в системе с Unix, Linux или другой разновидностью Unix, тогда можете превратить файлы кода Python в исполняемые программы, почти как поступали бы в случае программ, написанных на языке командной оболочки вроде `csh` или `ksh`. Такие файлы обычно называются *исполняемыми сценариями*. Применяя простую терминологию, исполняемые сценарии в стиле Unix – это всего лишь нормальные текстовые файлы, содержащие операторы Python, но обладающие двумя особенностями.

- Первая строка является специальной. Как правило, первая строка сценариев начинается с символов `# !`, за которыми следует путь к интерпретатору Python на компьютере.
- Сценарии обычно имеют права на выполнение. Как правило, файлы сценариев помечаются как исполняемые файлы для указания операционной системе, что они могут запускаться в качестве программ верхнего уровня. В системах Unix такой трюк достигается с помощью команды, подобной `chmod +x имя-файла.py`.

Рассмотрим пример для Unix-образных систем. Создайте в текстовом редакторе файл кода Python по имени `brian`:

```
#!/usr/local/bin/python
print('The Bright Side ' + 'of Life...')      # + означает конкатенацию строк
```

Специальная строка в начале файла сообщает системе, где находится интерпретатор Python. Формально первая строка является комментарием Python. Как упоминалось ранее, все комментарии в программах Python начинаются с символа `#` и распространяются до конца строки; они представляют собой место, куда вставляется дополнительная информация для людей, читающих ваш код. Но такой комментарий, как в первой строке файла `brian`, предназначен специальной для Unix, потому что командная оболочка использует его для выполнения программного кода в оставшейся части файла.

Также обратите внимание, что файл называется просто `brian` без суффикса `.py`, применяемого для файла модуля ранее. Добавление `.py` к имени не навредит (и может помочь запомнить, что это файл программы Python), но поскольку импортирование кода в данном файле другими модулями не планируется, имя файла несущественно. Если вы назначите файлу права на выполнение посредством команды оболочки `chmod +x brian`, то сможете запускать его в командной оболочке операционной системы, как если бы он был двоичным исполняемым файлом (чтобы показанная далее команда выполнилась, необходимо либо удостовериться в наличии `.`, т.е. текущего каталога, в переменной среды PATH, либо запускать файл сценария как `./brian`):

```
% brian  
The Bright Side of Life...
```

Трюк с поиском посредством `env` в Unix

В некоторых системах Unix можно избежать жесткого кодирования в файле сценария пути к интерпретатору Python за счет написания в первой строке специального комментария следующего вида:

```
#!/usr/bin/env python  
... здесь находится код сценария...
```

В данном случае программа `env` определяет местоположение интерпретатора Python согласно настройкам пути поиска в системе (в большинстве командных оболочек Unix за счет просмотра всех каталогов, перечисленных в переменной среды PATH). Подобная схема может быть более переносимой, т.к. нет нужды жестко кодировать путь, где установлен Python, в первой строке всех сценариев. Таким образом, даже если сценарии перемещаются на другой компьютер или Python переносится в новое местоположение, то придется обновить только переменную среды PATH, а не все сценарии.

При наличии доступа к `env` откуда угодно ваши сценарии будут выполняться независимо от того, где в системе установлен Python. На самом деле продемонстрированная форма с `env` в целом теперь является рекомендуемой вместо чего-то общего вроде `/usr/bin/python`, потому что на некоторых платформах Python может быть установлен в другом месте. Разумеется, это предполагает, что программа `env` везде находится в том же самом местоположении (`/sbin`, `/bin` или где-нибудь еще); если же нет, тогда о переносимости можно и не мечтать!

Запускающий модуль для Windows в версии Python 3.3: `#!` приходит в Windows

Примечание для пользователей Windows, работающих с *Python 3.2 и предшествующими версиями*: описанный здесь метод представляет собой трюк для Unix и на вашей платформе может не работать. Не переживайте; просто используйте показанный ранее базовый прием с командной строкой. Укажите имя файла в явной строке команды `python`¹:

¹ Как обсуждалось при исследовании командной строки, все последние версии Windows позволяют набирать в командной строке системы только имя файла `.py` – они применяют ассоциации расширений файлов для определения, что файл должен быть открыт с помощью Python (например, набор `brian.py` эквивалентен набору `python brian.py`). Такой режим командной строки похож по духу на `#!` из Unix, хотя он действует в масштабе всей системы, а не на основе файлов. Он также требует наличия у файла расширения `.py`: без него ассоциации расширений файлов не работают. В действительности некоторые программы в Windows способны интерпретировать первую строку с `#!` почти как в Unix (включая запускающий модуль для Windows из Python 3.3), но сама по себе командная строка в Windows просто игнорирует ее.

```
C:\code> python brian
The Bright Side of Life...
```

В данном случае специальный комментарий `#!` в начале сценария не нужен (если он присутствует, то Python его проигнорирует) и файл не нуждается в правах на выполнение. На самом деле, когда вы хотите запускать файлы переносимым образом между Unix и Microsoft Windows, ваша жизнь наверняка упростится, если для запуска программ вы всегда будете применять базовый подход с командной строкой, а не сценарии в стиле Unix.

Однако если вы используете *Python 3.3 или последующей версии*, либо имеете отдельно установленный запускающий модуль для Windows, тогда окажется, что строки `#!` в стиле Unix кое-что значат и в Windows. Упомянутый ранее в главе запускающий модуль для Windows помимо того, что предлагает описанную ранее команду `ru`, пытается произвести разбор строк `#!` с целью выяснения версии Python, которую нужно запустить для выполнения кода вашего сценария. Вдобавок он позволяет указывать номер версии в полной или частичной форме и распознает распространенные шаблоны Unix для этой строки, в том числе форму `/usr/bin/env`.

Механизм разбора `#!` запускающего модуля применяется, когда вы запускаете сценарии в командной строке посредством `ru` и когда дважды щелкаете на значках файлов Python (в случае чего `ru` запускается неявно ассоциациями расширений файлов). В отличие от Unix, чтобы это работало в Windows, выдавать файлам права на выполнение не требуется, т.к. аналогичные результаты обеспечивают ассоциации расширений файлов.

Например, первый сценарий из показанных ниже выполняется версией Python 3.X, а второй – версией Python 2.X (если число не указано явно, то по умолчанию запускающий модуль выбирает версию Python 2.X, если только вы не установили переменную среды `PYTHON`):

```
c:\code> type robin3.py
#!/usr/bin/python3
print('Run', 'away!...')      # Функция Python 3.X

c:\code> py robin3.py        # Выполнить файл согласно версии в строке)!
Run away!...

c:\code> type robin2.py
#!/python2
print 'Run', 'away more!...' # Оператор Python 2.X
c:\code> py robin2.py        # Выполнить файл согласно версии в строке)!
Run away more!...
```

Прием работает в дополнение к передаче номера версии в командной строке, что кратко демонстрировалось ранее для запуска интерактивной подсказки, но применимо и к запуску файла сценария:

```
c:\code> py -3.1 robin3.py    # Выполнить согласно аргументу командной строки
Run away!...
```

Совокупный эффект заключается в том, что запускающий модуль позволяет указывать версии Python на основе *файлов* и на основе *команд*, используя соответственно строки `#!` и аргументы командной строки. Такова очень краткая история запускающего модуля. Если вы работаете с Python 3.3 или последующей версией в Windows либо планируете перейти на нее в будущем, тогда я рекомендую ознакомиться с полной историей запускающего модуля в приложении Б второго тома.

Щелчки на значках файлов

Если вы не поклонник командной строки, то в большинстве случаев можете избежать ее применения, запуская сценарии Python с помощью щелчков на значках файлов, используя графические пользовательские интерфейсы для разработки и прибегая к другим схемам, которые зависят от платформы. Давайте бегло взглянем на первую из указанных альтернатив.

Основы щелчков на значках

Щелчки на значках в той или иной форме поддерживаются на большинстве платформ. Ниже приведена краткая сводка о том, как они могут выглядеть.

Щелчки на значках в системах Windows

В системах Windows реестр делает открытие файлов с помощью щелчков на значках очень легким. При установке Python применяет *ассоциации расширений* файлов Windows для автоматической регистрации себя в качестве программы, которая открывает файлы программ Python, когда на них совершается щелчок. В результате становится возможным запуск написанных программ Python простым щелчком (или двойным щелчком) кнопкой мыши на значках их файлов.

В частности, файл, на котором совершен щелчок, будет выполняться одной из двух программ Python в зависимости от его расширения и установленной версии Python. В Python 3.2 и предшествующих версиях файлы .ру выполняются программой python.exe с консольным окном (окном командной строки), а файлы .руw – программой pythonw.exe без консоли. Файлы байт-кода также выполняются упомянутыми программами, когда на них производится щелчок. Согласно приложению Б в Python 3.3 и последующих версиях те же самые роли исполняют программы ru.exe и ruw.exe запускающего модуля для Windows (а также в случае его отдельной установки), открывая файлы .ру и .руw соответственно.

Щелчки на значках в системах, отличных от Windows

В системах, отличных от Windows, скорее всего, вы сможете предпринимать аналогичные действия, но значки, схемы навигации в проводнике файлов и многое другое могут несколько отличаться. Скажем, в Mac OS X вы могли бы использовать для запуска PythonLauncher из папки MacPython (или Python N.M) в рамках папки Applications (Приложения), щелкая на Finder.

В Linux и других системах, основанных на Unix, вам может потребоваться зарегистрировать расширение .ру посредством графического пользовательского интерфейса проводника файлов, сделать сценарий исполняемым с применением строки #! или ассоциировать MIME-тип файла с приложением или командой путем редактирования файлов, установки программ либо использования других инструментов. Дополнительные сведения ищите в документации по проводнику файлов.

Другими словами, щелчки на значках в целом работают ожидаемым образом для имеющейся платформы, но при необходимости обращайтесь за деталями в стандартную документацию по Python.

Щелчки на значках в Windows

В целях иллюстрации продолжим работу с написанным ранее сценарием `script1.py`, который для удобства показан ниже:

```
# Первый сценарий Python
import sys                      # Загрузить библиотечный модуль
print(sys.platform)              # Возвести 2 в степень
print(2 ** 100)                  # Возвести 2 в степень
x = 'Spam!'
print(x * 8)                    # Повторить строку
```

Как вы уже видели, файл всегда можно запустить в командной строке системы:

```
C:\code> python script1.py
win32
1267650600228229401496703205376
Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

Тем не менее, щелчки на значках позволяют запустить файл, вообще ничего не набирая. Для этого вы должны отыскать значок файла в своей системе. В Windows 8 вы можете щелкнуть правой кнопкой мыши в левом нижнем углу экрана и открыть проводник файлов. В более ранних версиях Windows вы можете выбрать в меню Пуск пункт Компьютер (или Мой компьютер в XP). Существуют и другие способы открытия проводника файлов; открыв его, перейдите в свой рабочий каталог.

В данный момент вы должны видеть окно проводника файлов, подобное показанному на рис. 3.1 (здесь применяется Windows 8). Обратите внимание на то, как выглядят **значки** для файлов Python:

- файлы исходного кода имеют белый фон;
- файлы байт-кода имеют черный фон.

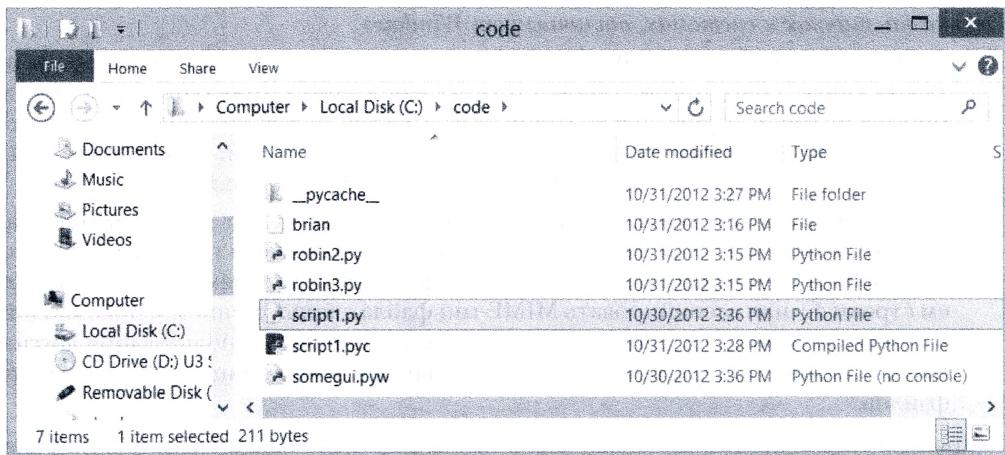


Рис. 3.1. В Windows файлы программ отображаются как значки в окне проводника файлов и могут запускаться по двойному щелчку кнопкой мыши (хотя в таком случае можно не увидеть вывод или сообщения об ошибках)

В предыдущей главе был создан файл байт-кода за счет импортирования в Python 3.1; в Python 3.2 и последующих версиях файлы байт-кода сохраняются в подкаталоге `_rucsache_`, который также присутствует на рис. 3.1. Обычно вы будете щелкать на файлах исходного кода (или запускать их по-другому), чтобы иметь дело с самыми последними изменениями, а не на файлах байт-кода — когда вы запускаете байт-код напрямую, то Python не проверяет файл исходного кода на предмет изменений. Для запуска файла просто дважды щелкните на значке `script1.py`.

Трюк с использованием функции `input` в Windows

К сожалению, в Windows результат щелчка на значке файла может не быть полностью удовлетворительным. На самом деле щелчок на имени файла рассматриваемого сценария приводит к озадачивающему быстрому мельканию на экране — точно не тот вид обратной связи, на который надеются будущие программисты на Python! Дело не в дефекте, а в том, каким образом версия Python для Windows обрабатывает вывод.

По умолчанию Python генерирует всплывающее консольное окно DOS (окно командной строки), обслуживающее ввод и вывод для файла, на котором совершен щелчок. Когда сценарий всего лишь производит вывод и завершается, как в настоящем примере, консольное окно открывается и текст отображается, но по завершении программы консольное окно закрывается и исчезает. Если только вы не проявите должную сноровку или ваш компьютер не является слишком медленным, то вывод вы вообще не увидите. Хотя это нормальное поведение, вряд ли оно будет тем, на что вы надеялись.

К счастью, проблему легко обойти. Если нужно, чтобы вывод сценария не исчезал после запуска щелчком на значке, тогда просто вызовите в самом конце сценария встроенную функцию `input` в версии Python 3.X (в версии Python 2.X взамен применийте имя `raw_input`: см. врезку “На заметку!” далее в главе). Например:

```
# Первый сценарий Python
import sys          # Загрузить библиотечный модуль
print(sys.platform) # Возвести 2 в степень
print(2 ** 100)
x = 'Spam!'
print(x * 8)        # Повторить строку
input()              # <== ДОБАВЛЕНО
```

Функция `input` читает и возвращает следующую строку из стандартного потока ввода с ожиданием, пока строка не будет готова. Конечным результатом в данном контексте окажется остановка сценария, за счет которой окно с выводом остается открытym вплоть до нажатия вами клавиши `<Enter>` (рис. 3.2).

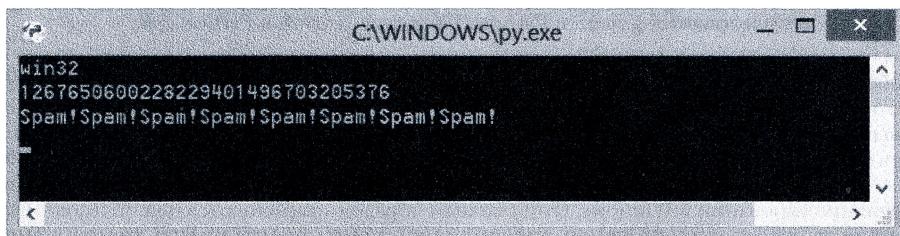


Рис. 3.2. После щелчка на значке программы в Windows вы сможете видеть ее вывод, если поместите в самый конец сценария вызов функции `input`. Но поступать так следует только в этом контексте!

Теперь, когда вы ознакомились с трюком, имейте в виду, что он обычно требуется только для Windows, только в случае, если сценарий выводит текст и завершается, и только при запуске сценария путем щелчка на значке его файла. Вы должны добавлять вызов функции `input` в конец своих сценариев верхнего уровня тогда и только тогда, когда все указанные условия удовлетворены. Нет никаких причин добавлять данный вызов в любых других контекстах, таких как сценарии, запускаемые в командной строке или в графическом пользовательском интерфейсе IDLE (если только вы не испытываете неумеренное пристрастие к нажиманию клавиши `<Enter>!`)². Это может выглядеть очевидным, но является еще одним частым источником проблем в реальных классах.

Прежде чем двигаться дальше, следует отметить, что используемый здесь вызов `input` является входным эквивалентом применения функции `print` (оператора в Python 2.X) для вывода. Функция `input` представляет собой простейший способ чтения пользовательского ввода и более универсальна, чем вытекает из рассмотренного примера. Функция `input`:

- дополнительно принимает строку, которая будет выводиться в качестве приглашения к вводу (например, `input('Press Enter to exit')`);
- возвращает сценарию прочитанный текст в виде строки (например, `nextinput = input()`);
- поддерживает перенаправление входного потока на уровне командной оболочки системы (например, `python spam.py < input.txt`) точно так же, как оператор `print` позволяет делать это для вывода.

Позже мы будем использовать функцию `input` более сложными способами; скажем, в главе 10 она будет применяться в интерактивном цикле. Пока что она помогает видеть вывод простых сценариев, которые запускаются щелчком на значках их файлов.



Примечание, касающееся нестыковки версий. Если вы работаете в Python 2.X, тогда используйте в коде `raw_input()` вместо `input()`. В версии Python 3.X первая функция была переименована и стала второй. Формально в Python 2.X также имеется функция `input`, но она *оценывает* строки, как если бы они представляли собой программный код, набранный в сценарии, и в этом контексте работать не будет (пустая строка приводит к ошибке). Функция `input` в Python 3.X (и `raw_input` в Python 2.X) просто возвращает введенный текст как строку символов без оценки. Чтобы смоделировать функцию `input` из Python 2.X в Python 3.X, применяйте `eval(input())`. Однако имейте в виду, что поскольку это выполняет введенный тест, как если бы он был *программным кодом*, могут возникнуть последствия в плане безопасности, которые мы здесь почти полностью игнорируем, не говоря даже о необходимости доверия к источнику вводимого текста. При отсутствии такого доверия придерживайтесь использования `input` в Python 3.X и `raw_input` в Python 2.X.

² И наоборот, можно также полностью подавить всплывающее консольное окно (окно командной строки) для файлов, на которых совершен щелчок в среде Windows, когда вы *не* хотите видеть выводимый текст. Файлы с именами, заканчивающимися расширением `.ruw`, будут отображать только окна, которые созданы сценарием, но не стандартное консольное окно. Файлы `.ruw` – это просто файлы исходного кода `.py`, обладающие таким специальным поведением при функционировании в Windows. Они главным образом применяются для пользовательских интерфейсов, написанных на Python, которые самостоятельно строят окна, часто в сочетании с различными приемами сохранения вывода и сообщений об ошибках в файлы. Как предполагалось ранее, это достигается, когда Python устанавливается за счет ассоциирования специального исполняемого файла (`pythonw.exe` в Python 3.2 и предшествующих версиях и `ruw.exe` в версии Python 3.3) для открытия файлов `.ruw`, когда на их именах делается щелчок.

Другие ограничения, связанные со щелчками на значках

Даже когда применяется трюк с `input`, описанный в предыдущем разделе, щелчки на значках файлов не обходятся без возникновения опасностей. Вы также можете не получить сообщения об ошибках от Python. Если ваш сценарий генерирует ошибку, то текст ее сообщения выводится во всплывающем консольном окне, которое затем немедленно исчезает! Хуже того, добавление в файл вызова `input` на этот раз не поможет, потому что сценарий прервет работу вероятно задолго до достижения данного вызова. Другими словами, вы не будете в состоянии сказать, что именно пошло не так.

Из обсуждения исключений позже в книге вы узнаете, что возможно написание кода для перехвата, обработки и восстановления после ошибок, чтобы они не приводили к прекращению работы программ. При обсуждении оператора `try` далее в книге будет показан альтернативный способ недопущения закрытия консольного окна в случае возникновения ошибок. Когда будут рассматриваться операции `print`, вы также научитесь перенаправлять выводимый текст в файлы для исследования в более позднее время. Тем не менее, за исключением такой поддержки в коде сообщения об ошибках и вывод теряются для программ, запускаемых щелчком на их значках.

Из-за описанных ограничений щелчки на значках, пожалуй, лучше всего считать способом запуска программ лишь после их отладки или оснащения инструментами для записи их вывода в файл и перехвата и обработки любых важных ошибок. Рекомендуется, особенно поначалу, использовать другие приемы, такие как командная строка системы и IDLE (обсуждается в разделе “Пользовательский интерфейс IDLE” далее в главе), что позволит вам видеть генерируемые сообщения об ошибках и обычный вывод, не прибегая к написанию добавочного кода.

Импортирование и повторная загрузка модулей

До сих пор я говорил об “импортировании модулей”, не вдаваясь в подробности, что этот термин означает. Модули и более крупные программные архитектуры будут исследованы в части V, но поскольку импортирование также является способом запуска программ, в текущем разделе представлены основы модулей, которых должно быть достаточно для начала.

Основы импортирования и повторной загрузки

Выражаясь простыми словами, каждый файл исходного кода Python, чье имя заканчивается расширением `.py`, является модулем. Чтобы сделать файл модулем, никакого специального кода или синтаксиса не требуется: модулем будет любой файл такого рода. Другие файлы могут получать доступ к элементам, определенным в модуле, *импортируя* этот модуль – операции импорта по существу загружают другой файл и предоставляют доступ к его содержимому. Содержимое модуля делается доступным внешнему миру через его атрибуты (термин, который объясняется в следующем разделе).

Такая модель служб на основе модулей оказывается основной идеей *программной архитектуры* в Python. Более крупные программы обычно принимают форму множества файлов модулей, которые импортируют инструменты из других файлов модулей. Один из модулей предназначен быть главным файлом либо файлом *верхнего уровня*, или “сценарием” – файлом, запускаемым для старта целой программы, которая выполняется строкой за строкой обычным образом. Ниже данного уровня все модули импортируют другие модули.

Позже в книге мы углубимся в такие архитектурные детали. В настоящей главе нас больше всего интересует тот факт, что операции импорта в качестве финального шага *выполняют* код в загружаемом файле. Из-за этого импортирование файла является еще одним способом его запуска.

Например, если вы начнете интерактивный сеанс (из командной строки системы или как-то иначе), тогда сможете запустить созданный ранее файл `script1.py` с помощью простого импорта (сначала удалите строку с вызовом `input`, добавленную в предыдущем разделе, а то придется без веской причины нажимать `<Enter>`):

```
C:\code> C:\python33\python
>>> import script1
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam!
```

Сценарий работает, но по умолчанию только однократно на сеанс (в действительности на *процесс* – выполнение программы). После первого импорта дальнейшие операции импорта ничего не делают, даже если вы измените и сохраните файл исходного кода модуля в другом окне:

...Изменение `script1.py` в окне текстового редактора для вывода 2 ** 16...

```
>>> import script1
>>> import script1
```

Так было задумано; импорт – слишком затратная операция, чтобы повторять ее более одного раза в файле на выполнение программы. В главе 22 вы узнаете, что операции импорта обязаны искать файлы, компилировать их в байт-код и выполнять код.

Если на самом деле необходимо заставить Python выполнить файл снова в том же самом сеансе, не останавливая его и не перезапуская, то взамен понадобится вызвать функцию `reload`, доступную в стандартном библиотечном модуле `imp` (она является просто встроенной функцией в Python 2.X, но не в Python 3.X):

```
>>> from imp import reload # Должна загружаться из модуля в Python 3.X (только)
>>> reload(script1)
win32
65536
Spam! Spam! Spam! Spam! Spam! Spam!
<module 'script1' from '.\\script1.py'>
<модуль script1 из .\\script1.py>
>>>
```

Оператор `from` просто копирует имя из модуля (вскоре мы обсудим это подробнее). Сама функция `reload` загружает и выполняет текущую версию кода файла, подхватывая изменения, если он был модифицирован и сохранен в другом окне.

В итоге у вас появляется возможность редактировать и подхватывать новый код на лету внутри текущего интерактивного сеанса Python. Скажем, в данном сеансе между моментом выполнения первого оператора `import` и моментом вызова `reload` второй оператор `print` сценария `script1.py` был изменен в другом окне, чтобы выводить `2 ** 16` – отсюда и другой результат.

Функция `reload` ожидает передачи имени объекта уже загруженного модуля, а потому вам нужно иметь успешно импортированный модуль, прежде чем перезагружать его (если операция импорта сообщает об ошибке, то вы не сможете перезагрузить и должны импортировать модуль заново). Обратите внимание, что `reload` также ожи-

дает, что имя объекта модуля помещено в круглые скобки, тогда как `import` – нет. Причина в том, что `reload` – функция, которая *вызывается*, а `import` – оператор.

Вот почему вы обязаны передавать `reload` имя модуля как аргумент в круглых скобках и получать обратно добавочную строку вывода при перезагрузке – последняя строка вывода является просто отображаемым представлением возвращаемого значения вызова `reload`, т.е. объекта модуля Python. Вы узнаете больше о работе с функциями в главе 16, а пока, когда вы имеете дело с “функцией”, не забывайте, что для ее вызова требуются круглые скобки.



Примечание, касающееся нестыковки версий. В версии Python 3.X встроенная функция `reload` была перемещена в стандартный библиотечный модуль `imp`. Она по-прежнему перезагружает файлы, как и ранее, но для использования вы должны ее импортировать. В Python 3.X укажите `import imp` и применяйте `imp.reload(M)` или `from imp import reload` и используйте `reload(M)`, как показано здесь. Мы обсудим операторы `import` и `from` в следующем разделе и более формально позже в книге.

Если вы работаете в версии Python 2.X, тогда `reload` доступна как встроенная функция и какое-либо импортование не требуется. В версиях Python 2.6 и 2.7 функция `reload` доступна в *обеих* формах, как встроенная и как функция модуля, чтобы содействовать переходу на Python 3.X. Другими словами, перезагрузка в Python 3.X доступна, но необходима дополнительная строка кода для извлечения `reload`.

Отчасти причиной такого перемещения функции в Python 3.X вероятно были хорошо известные проблемы, связанные с операторами `reload` и `from`, с которыми вы ознакомитесь в следующем разделе. Говоря кратко, загружаемые с помощью `from` имена не обновлялись напрямую `reload`, но имена, доступ к которым осуществлялся посредством оператора `import`, обновлялись. Если после `reload` имена не выглядят изменившимися, тогда попробуйте взамен применять `import` и ссылки на имена модуля `.атрибут`.

Дополнительная история о модулях: атрибуты

Импортование и перезагрузка предоставляют естественный вариант запуска программ, потому что операторы импорта выполняют файлы в качестве своего последнего шага. Но как вы узнаете в части V, в более широкой схеме модули исполняют роль библиотек инструментов. Однако базовая идея прямолинейна: по большей части модуль представляет собой лишь пакет имен переменных, известный под названием *пространство имен*, и имена внутри такого пакета называются *атрибутами*. Атрибут – это просто имя переменной, которое прикрепляется к специальному объекту (вроде модуля).

Выражаясь более конкретно, импортирующие модули получают доступ ко всем именам, определенным на верхнем уровне файла модуля. Имена обычно назначаются инструментам, экспортруемым модулем (функциям, классам, переменным и т.д.), которые предназначены для использования в других файлах и программах. Снаружи имена файла модуля можно извлечь с помощью двух операторов Python, `import` и `from`, а также вызова `reload`.

В целях иллюстрации создайте в своем текстовом редакторе односторонний файл модуля Python по имени `myfile.py` со следующим содержимым, поместив его в рабочий каталог:

```
title = "The Meaning of Life"
```

Такой модуль Python вполне можно считать одним из самых простых в мире (он содержит единственный оператор присваивания), но он достаточен в плане демонстрации. Когда файл `myfile.py` импортируется, его код выполняется для генерации атрибута модуля, т.е. оператор присваивания создает переменную и атрибут модуля по имени `title`.

Получить доступ к атрибуту `title` модуля в других компонентах можно двумя способами. Вы можете загрузить модуль целиком посредством оператора `import` и затем уточнить имя модуля именем атрибута, чтобы извлечь его (обратите внимание, что здесь вы разрешаете интерпретатору выводить автоматически):

```
% python          # Запустить Python
>>> import myfile    # Выполнить файл; загрузить модуль целиком
>>> myfile.title    # Использовать имя атрибута: . служит для уточнения
'The Meaning of Life'
```

В общем случае синтаксис выражения с точкой `объект.атрибут` позволяет извлечь любой атрибут, прикрепленный к любому объекту, и является одной из самых распространенных операций в коде Python. Мы применяем ее для доступа к строковой переменной `title` внутри модуля `myfile` – другими словами, `myfile.title`.

В качестве альтернативы извлекать (в действительности копировать) имена из модуля можно с помощью операторов `from`:

```
% python          # Запустить Python
>>> from myfile import title    # Выполнить файл; копировать его имена
>>> title          # Использовать имя напрямую: уточнять не нужно
'The Meaning of Life'
```

Как вы увидите позже, оператор `from` похож на `import`, но с добавочным назначением имен в импортируемом компоненте. Формально `from` копирует *атрибуты* модуля, так что они становятся просто *переменными* на принимающей стороне – таким образом, вы можете ссылаться на импортированную строку как на `title` (переменную), а не `myfile.title` (ссылка на атрибут)³.

³ Обратите внимание, что в `import` и `from` имя модуля указывается в виде `myfile` без расширения `.py`. В части V вы узнаете, что когда интерпретатор Python ищет фактический файл, ему известно о включении расширения в свою процедуру поиска. Вы обязаны включать расширение `.py` в командную строку системы, но не в операторы `import`.

Независимо от того, применяете вы `import` или `from` для обращения к операции импорта, операторы в файле модуля `myfile.py` выполняются, а импортирующий компонент (в данном случае интерактивная подсказка) получает доступ к именам, определенным на верхнем уровне файла. В этом простом примере существует только одно такое имя – переменная `title`, которой присвоена строка. Но концепция будет более полезной, когда вы начнете определять в своих модулях объекты вроде функций и классов: объекты подобного рода становятся многократно используемыми *программными компонентами*, к которым можно обращаться по именам из одного или большего числа клиентских модулей.

На практике файлы модулей обычно определяют более одного имени для применения внутри и вне файлов. Вот пример определения трех имен:

```
a = 'dead'      # Определить три атрибута
b = 'parrot'    # Экспортировать в другие файлы
c = 'sketch'
print(a, b, c)  # Так же использовать в этом файле (в Python 2.X: print a, b, c)
```

В файле `threenames.py` определяются три переменные, а потому для внешнего мира он генерирует три атрибута. Он также применяет эти три переменные в операторе `print` версии Python 3.X, как можно увидеть в случае его запуска как файла верхнего уровня (в Python 2.X использование `print` несколько отличается, так что для получения точно такого же вывода опустите внешние круглые скобки; более полное объяснение приведено в главе 11):

```
% python threenames.py  
dead parrot sketch
```

Весь код из данного файла выполняется обычным образом при первом его импортировании где-то в другом месте посредством `import` или `from`. Клиенты файла, которые применяют `import`, получают модуль с атрибутами, в то время как клиенты, использующие `from`, получают копии имен из файла:

```
% python  
>>> import threenames      # Захватить весь модуль: здесь он выполняется  
dead parrot sketch  
>>>  
>>> threenames.b, threenames.c      # Получить доступ к атрибутам  
('parrot', 'sketch')  
>>>  
>>> from threenames import a, b, c # Копировать множество атрибутов  
>>> b, c  
('parrot', 'sketch')
```

Результаты здесь выводятся в круглых скобках, поскольку на самом деле они являются *кортежами* – разновидность объекта, создаваемого с применением запятых во входных данных (кортежи рассматриваются в следующей части книги); пока можете это благополучно проигнорировать.

Как только вы начинаете писать код модулей с множеством имен, становится полезной встроенная функция `dir` – вы можете использовать ее для извлечения списка всех имен, доступных внутри модуля. Приведенный ниже вызов возвращает список строк Python в квадратных скобках (изучение списков начнется в следующей главе):

```
>>> dir(threenames)  
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'a', 'b', 'c']
```

Содержимое списка было отредактировано, т.к. оно варьируется в зависимости от версии Python. Здесь важно отметить, что когда функция `dir` вызывается с именем импортированного модуля в круглых скобках, она возвращает все атрибуты внутри модуля. Ряд возвращаемых имен вы получаете “бесплатно”: имена с ведущими и завершающими двойными подчеркиваниями (`__X__`) являются встроенными именами, которые всегда предварительно определены Python и имеют специальный смысл для интерпретатора, но в настоящий момент они не важны. Переменные нашего кода, определенные присваиваниями (`a`, `b` и `c`), в результате вызова функции `dir` отображаются последними.

Модули и пространства имен

Импортирование модулей позволяет выполнять файлы кода, но позже в книге будет показано, что модули представляют собой важнейшую программную структуру в программах Python и одну из главных ключевых концепций в языке.

Как уже объяснялось, программы Python состоят из множества файлов модулей, связанных вместе операторами `import`, и каждый файл модуля является пакетом пе-

ременных, т.е. *пространством имен*. Не менее важно и то, что каждый модуль представляет собой *автономное* пространство имен: один файл модуля не может видеть имена, определенные в другом файле, до тех пор, пока явно не импортирует другой файл. Из-за этого модули содействуют минимизации *конфликтов имен* в коде — поскольку каждый файл является автономным пространством имен, имена в одном файле не могут конфликтовать с именами в другом файле, даже если они написаны одинаково.

Фактически модули — один из немногих способов, которыми Python делает все возможное, чтобы упаковать ваши переменные в группы во избежание конфликтов имен. Позже в книге мы еще будем обсуждать модули и другие конструкции пространств имен, в том числе локальные области видимости, определяемые классами и функциями. А пока модули пригодятся в качестве способа многократного выполнения кода без необходимости в его повторном наборе и предотвращения непредумышленного замещения имен, определенных в файлах.



`import` или `from`. Важно отметить, что оператор `from` в некотором смысле аннулирует цель разделения на пространства имен, преследуемую модулями. Поскольку `from` копирует переменные из одного файла в другой, он безо всякой предупреждения может привести к переписыванию одинаково именованных переменных в файле, в который производится импорт. Такое действие по существу объединяет вместе пространства имен, во всяком случае, с точки зрения копируемых переменных.

По указанной причине некоторые рекомендуют всегда применять `import`, а не `from`. Тем не менее, я не буду заходить настолько далеко; оператор `from` не только требует меньшего клавиатурного набора (ценное качество при работе в интерактивной подсказке), но и описанная выше проблема на практике возникает довольно редко. Вдобавок это именно то, что вы контролируете, перечисляя желаемые переменные в операторе `from`. До тех пор, пока вы осознаете, что им будут присвоены значения в целевом модуле, такое действие не более опасно, чем операторы присваивания — еще одно средство, которым вы непременно будете пользоваться!

Замечания по использованию: `import` и `reload`

Почему-то когда люди узнают о выполнении файлов с применением операторов `import` и `reload`, то многие ограничиваются только ими и забывают о других вариантах запуска, которые всегда выполняют текущую версию кода (например, щелчки на значках, пункты меню IDLE и командная строка системы). Однако такой подход может быстро привести к путанице — вам необходимо помнить при импортировании, можете ли вы перезагружать, не забывать использовать круглые скобки в случае вызова `reload` (только) и помнить о применении в первую очередь `reload` для получения текущей версии выполняемого кода. Кроме того, перезагрузка не обладает переходным характером; перезагрузка воздействует только на указанный модуль, но не на модули, которые он может импортировать, поэтому иногда приходится перезагружать множество файлов.

Из-за описанных сложностей (и других, которые мы исследуем позже, в том числе проблемы `reload/from`, кратко упомянутой во врезке “На заметку!” ранее в главе) пока что в целом неплохо не поддаваться соблазну запускать файлы с помощью импортирования и перезагрузки. Например, описанный в следующем разделе пункт меню `Run`⇒`Run Module` (Выполнить⇒Выполнить модуль) в IDLE предоставляет более простой и менее подверженный ошибкам способ запуска файлов и всегда выполняет

текущую версию кода. Командная строка системы предлагает похожие преимущества. В случае использования любых таких приемов необходимость в применении `reload` исчезает.

Вдобавок вы можете столкнуться с проблемами, если будете прямо сейчас использовать модули необычными способами. Скажем, если вас интересует импортирование файла модуля, который хранится не в том каталоге, где вы работаете, тогда придется переключиться на главу 22 и почитать о *путях поиска модулей*. Пока что, когда нужно импортирование, старайтесь хранить все файлы в рабочем каталоге во избежание затруднений³.

Тем не менее, импортирование и перезагрузка оказались популярной методикой тестирования в классах Python, и вы также можете отдать предпочтение такому подходу. Однако, как обычно, если вы обнаруживаете, что уперлись в стену, то не пытайтесь ее пробить!

Использование `exec` для выполнения файлов модулей

Строго говоря, для выполнения кода, хранящегося в файлах модулей, существует больше способов, чем было представлено до сих пор. Например, вызов встроенной функции `exec(open('module.py').read())` является еще одним приемом запуска файлов из интерактивной подсказки без импортирования и последующей перезагрузки. Каждый такой вызов `exec` выполняет *текущую* версию кода, прочитанную из файла, не требуя перезагрузки в более позднее время (`script1.py` находится в том состоянии, в каком мы его оставили после перезагрузки в предыдущем разделе):

```
% python
>>> exec(open('script1.py').read())
win32
65536
Spam!Spam!Spam!Spam!Spam!Spam!Spam!
...Изменение script1.py в окне текстового редактора для вывода 2 ** 32...
>>> exec(open('script1.py').read())
win32
4294967296
Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

Эффект от вызова `exec` похож на результат оператора `import`, но в действительности не приводит к импортированию модуля. По умолчанию каждый раз, когда вы вызываете `exec` подобным образом, он выполняет код из файла заново, как будто он был вставлен в то место, где находится вызов `exec`. Из-за этого `exec` не требует перезагрузки модуля после внесения изменений в файл — он пропускает обычную логику импортирования модуля.

С другой стороны, поскольку вызов `exec` работает, как если бы вы поместили код на его место, подобно упомянутому ранее оператору `from`, потенциально он способен молча переписать переменные, которые используются в текущий момент. Скажем,

³ Если вы слишком любознательны, чтобы ждать до главы 22, то вот вам краткий пересказ: Python ищет импортируемые модули в каждом каталоге, перечисленном в `sys.path` — список Python строк с именами каталогов в модуле `sys`, который инициализируется содержимым переменной окружения `PYTHONPATH` плюс набором стандартных каталогов. Когда необходимо импортировать из каталога, отличающегося от того, где вы работаете, то этот каталог обычно должен присутствовать в `PYTHONPATH`. За дополнительными деталями обращайтесь в главу 22 и приложение А второго тома.

в нашем сценарии `script1.py` происходит присваивание переменной по имени `x`. Если такое имя применяется в месте вызова `exec`, тогда значение `x` заменяется:

```
>>> x = 999
>>> exec(open('script1.py').read())      # По умолчанию код выполняется
                                         # в этом пространстве имен
... тот же самый вывод...
>>> x                               # Здесь могут быть перезаписаны его присваивания
'Spam!'
```

По контрасту с этим базовый оператор `import` выполняет файл только один раз на процесс и делает файл отдельным пространством имен модуля, так что его присваивания не изменят переменные в вашей области видимости. Платой за разделение пространств имен модулей будет необходимость в перезагрузке после внесения изменений.



Примечание, касающееся нестыковки версий. В добавок к поддержке вызова в форме `exec(open('module.py'))` версия Python 2.X также включает встроенную функцию `execfile('module.py')`. Обе функции автоматически читают содержимое файла и эквивалентны вызову в форме `exec(open('module.py').read())`, который сложнее, но выполняется как в Python 2.X, так и в Python 3.X.

К сожалению, ни одна из двух более простых форм, имеющихся в Python 2.X, не доступна в Python 3.X, поэтому вы должны понимать их обе. Фактически с формой `exec` в Python 3.X связано настолько много клавиатурного набора, что добрым советом, вероятно, будет просто отказ от ее использования – запускать файлы обычно легче из командной строки системы или через меню IDLE, как будет описано в следующем разделе.

Дополнительные сведения об интерфейсах файлов, применяемых формой `exec` в Python 3.X, приведены в главе 9. Больше информации о функции `exec` и связанных с ней функциях `eval` и `compile` можно найти в главах 10 и 25.

Пользовательский интерфейс IDLE

До сих пор вы видели, каким образом выполнять код Python с помощью интерактивной подсказки, командной строки системы, сценариев в стиле Unix, щелчков на значках, импортирования модулей и вызовов `exec`. Если вас интересует что-то визуальное, то *IDLE* предоставляет графический пользовательский интерфейс для разработки на Python, являясь стандартной и бесплатной частью системы Python. Обычно IDLE называют *интегрированной средой разработки* (*integrated development environment* – IDE), т.к. она увязывает вместе разнообразные задачи разработки в единственном представлении⁴.

Выражаясь кратко, среда IDLE – это настольный графический пользовательский интерфейс, который позволяет редактировать, выполнять, просматривать и отлаживать программы Python, используя единый интерфейс. Она функционирует на большинстве платформ Python, включая Microsoft Windows, X Window (для Linux и разновидностей Unix) и Mac OS (Classic и OS X).

⁴ IDLE – официально считается искаженной аббревиатурой IDE, но на самом деле названа в честь члена группы “Монти Пайтон” Эрика Айдла (Eric Idle). Дополнительную информацию ищите в главе 1.

Для многих IDLE представляет собой легкую в применении замену работе в командной строке, менее подверженную ошибкам альтернативу щелчкам на значках и великолепный способ для начинающих приступить к редактированию и выполнению кода. Вы утратите определенный контроль над некоторыми вещами, но обычно это становится важным только позже в карьере разработчика на Python.

Детали запуска IDLE

Многие читатели должны иметь возможность использовать среду IDLE незамедлительно, поскольку в наши дни она является стандартным компонентом в Mac OS X и в большинстве версий Linux, а также устанавливается автоматически при установке стандартного Python в Windows. Тем не менее, с каждой платформой связаны свои особенности, и потому прежде чем открывать этот графический пользовательский интерфейс, полезно ознакомиться с приведенными ниже советами.

Формально IDLE представляет собой программу Python, которая при создании своих окон применяет инструментальный набор для построения графических пользовательских интерфейсов Tkinter (Tkinter в Python 2.X) из стандартной библиотеки. Такое решение делает среду IDLE переносимой (она работает одинаково на всех крупных настольных платформах), но оно также означает необходимость наличия поддержки Tkinter для Python, чтобы можно было использовать IDLE. Поддержка Tkinter для Python является стандартной в Windows, Mac OS и Linux, но в некоторых системах сопряжена с удовлетворением нескольких условий, а запуск может варьироваться в зависимости от платформы.

- В Windows 7 и предшествующих версиях запускать среду IDLE легко – она всегда присутствует после установки Python и имеет пункт в меню для Python, доступном через кнопку Пуск (см. рис. 2.1). Ее также можно выбирать в контекстном меню, открывающемся по щелчку правой кнопкой мыши на значке программы Python, и запускать щелчком на значках файлов `idle.pyw` или `idle.py`, которые находятся в подкаталоге `idlelib` каталога `Lib` системы Python. В таком режиме IDLE является сценарием Python, запускаемым по щелчку, который расположен в `C:\Python33\Lib\idlelib`, `C:\Python27\Lib\idlelib` или похожем месте и допускает создание ярлыка для ускоренного доступа.
- В Windows 8 и последующих версиях ищите IDLE внутри плиток экрана Start, выполняя поиск `idle`, просматривая окно All apps или применяя проводник файлов для поиска упоминавшегося ранее файла `idle.py`. Можете также создать ярлык для ускоренного доступа.
- В Mac OS X все, что требуется для IDLE, присутствует в виде стандартных компонентов операционной системы. Среда IDLE должна быть доступной для запуска в папке Applications внутри папки MacPython (или Python N.M). Важно отметить, что некоторые версии Mac OS X могут требовать установки обновленной поддержки Tkinter из-за тонких зависимостей версий, которые здесь не рассматриваются; детали ищите на странице загрузки веб-сайта python.org.
- В Linux на сегодняшний день среда IDLE обычно представлена как стандартный компонент. Она может принимать форму исполняемого файла или сценария `idle` в пути поиска; введите `idle` в окне командной оболочки для проверки. На одних компьютерах может требоваться установка среды IDLE (советы ищите в приложении А тома 2), а на других – запуск сценария верхнего уровня IDLE в командной строке либо с помощью щелчка на значке: `idle.py` из подкаталога `idlelib` каталога `/usr/lib` системы Python (точное местоположение определяется посредством `find`).

Поскольку IDLE – всего лишь сценарий Python в пути поиска модулей внутри стандартной библиотеки, обычно вы также можете запускать его на любой платформе и из любого каталога, набирая показанную ниже команду в окне командной оболочки системы (например, в окне командной строки в Windows), хотя вам следует обратиться к приложению А второго тома за описанием флага `-m` при запуске Python и прочитать часть V книги, где объясняется требующийся здесь синтаксис . работы с пакетами (пока достаточно принять сказанное на веру):

```
c:\code> python -m idlelib.idle # Запустить idle.py из пакета  
# в пути поиска модулей
```

Чтобы ознакомиться с проблемами при установке и замечаниями по использованию для Windows и других платформ, просмотрите приложение А и раздел стандартного руководства по Python, посвященный установке и применению Python на имеющейся у вас платформе.

Базовое использование IDLE

Давайте рассмотрим пример. На рис. 3.3 показано главное окно оболочки Python сразу после запуска IDLE в Windows, где выполняется интерактивный сеанс (обратите внимание на приглашение >>>). Это похоже на все интерактивные сеансы – набираемый здесь код выполняется немедленно – и служит инструментом для тестирования и экспериментирования.

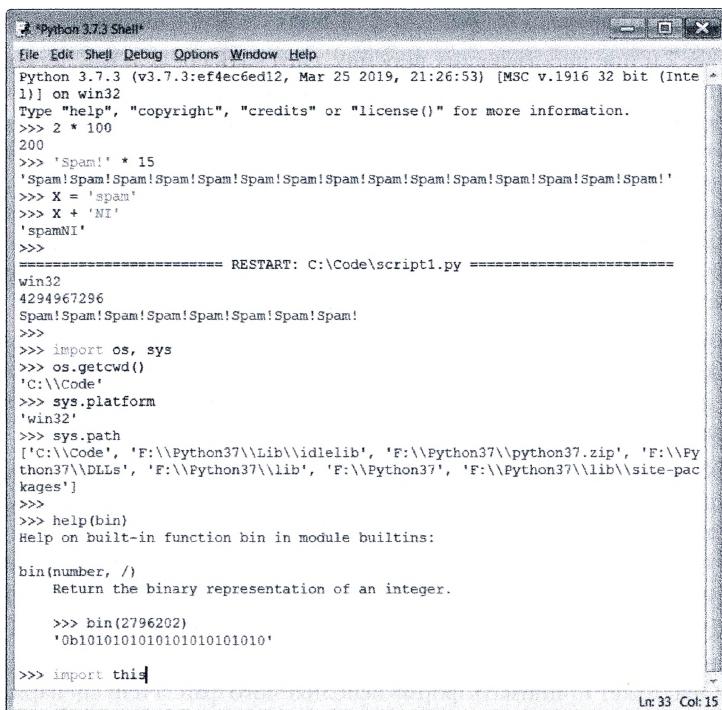


Рис. 3.3. Главное окно оболочки Python графического пользователяского интерфейса для разработки IDLE, функционирующее под управлением Windows. Применяйте меню File (Файл), чтобы создать (пункт New Window (Новое окно)) либо изменить (пункт Open... (Открыть...)) файл исходного кода; используйте меню Run (Выполнить) окна текстового редактора для выполнения кода в этом окне (пункт Run Module (Выполнить модуль))

Среда IDLE предлагает знакомые меню с клавиатурными комбинациями для большинства операций. Чтобы создать новый файл сценария в IDLE, применяйте пункт меню **File**⇒**New Window**: в главном окне оболочки выберите в раскрывающемся меню **File** пункт **New Window** (**New File** (Новый файл) в версиях 3.3.8 и 2.7.6) для открытия окна текстового редактора, в котором вы можете набирать, сохранять и выполнять код файла. Используйте пункт меню **File**⇒**Open...**, чтобы открыть окно текстового редактора с кодом существующего файла для редактирования и выполнения.

Хотя в черно-белой печатной книге этого не видно, среда IDLE применяет для кода, набираемого в главном окне и всех окнах текстового редактора, цветовую раскраску синтаксиса — ключевые слова отображаются одним цветом, литералы другим и т.д. В итоге вам легче получить общее представление о компонентах в коде (и даже проще выявить ошибки — например, все строки представлены одним цветом).

Чтобы выполнить файл кода, который вы отредактировали в IDLE, используйте пункт меню **Run**⇒**Run Module** в окне текстового редактора файла, т.е. выберите в раскрывающемся меню **Run** пункт **Run Module** (или нажмите клавиатурную комбинацию, указанную в пункте меню). Python сообщит вам, что сначала нужно сохранить файл, если вы его изменили после открытия или последнего сохранения и не сохранили изменения — распространенная ситуация, когда приходится интенсивно писать код.

При таком запуске вывод сценария и любые сообщения об ошибках, которые он может генерировать, отображаются в главном окне интерактивного сеанса (окне командной оболочки Python). Например, на рис. 3.3 три строки после строки с **RESTART** (Перезапуск) близко к середине окна отображают выполнение файла `script1.py`, открытого в отдельном окне редактора. Сообщение **RESTART** указывает на то, что процесс пользовательского кода был перезапущен, чтобы выполнить отредактированный сценарий, и служит разделителем для вывода сценария (оно не появится, если среда IDLE запущена без подпроцесса пользовательского кода — вскоре вы узнаете об этом больше).

Удобные функциональные возможности IDLE

Подобно большинству графических пользовательских интерфейсов лучший способ изучения среды IDLE предусматривает ее пробное применение, но некоторые ключевые моменты, связанные с использованием, не вполне очевидны. Например, если вы хотите повторить предшествующие команды в главном интерактивном окне IDLE, то можете применить клавиатурную комбинацию **<Alt+P>** для прокручивания хронологии команд назад и **<Alt+N>** для ее прокручивания вперед (на компьютерах Mac попробуйте вместо них **<Ctrl+P>** и **<Ctrl+N>**). Введенные ранее команды снова отобразятся и могут быть отредактированы и повторно запущены.

Повторять команды можно также путем перемещения на них курсора с последующим щелчком и нажатием **<Enter>** для вставки их текста в строку с приглашением к вводу либо использования стандартных операций вырезания и вставки, хотя такие приемы обычно требуют больше шагов (и временами срабатывают случайно). За пределами IDLE повторять команды в интерактивном сеансе на компьютере Windows можно с помощью клавиш со стрелками.

Кроме хронологии команд и цветовой раскраски синтаксиса среда IDLE обладает и другими удобными функциональными возможностями:

- автоматическое добавление и удаление отступов для кода Python в редакторе (клавиша **забоя** возвращает назад на один уровень);
- автоматическое завершение слов во время набора, вызываемое нажатием **<Tab>**;

- всплывающая подсказка для вызова функции после набора открывающей круглой скобки;
- всплывающий список выбора атрибутов объекта при наборе . после имени объекта с последующей паузой или нажатием <Tab>.

Некоторые средства работают не на всех платформах, а некоторые могут быть сконфигурированы или отключены, если они создают препятствия вашему персональному стилю написания кода.

Расширенные инструменты IDLE

Кроме базовых функций редактирования и выполнения и удобных средств, описанных в предыдущем разделе, среда IDLE предоставляет расширенные инструменты, такие как *графический отладчик* программ и *инспектор объектов*. Отладчик IDLE активизируется с помощью меню Debug (Отладка), а инспектор объектов – посредством меню File. Инспектор объектов позволяет просматривать классы и переходить через путь поиска модулей к файлам и объектам в файлах. Щелчок на имени файла или объекта приводит к открытию соответствующего исходного кода в окне текстового редактора.

Чтобы инициировать отладку IDLE, необходимо выбрать пункт меню Debug⇒ Debugger (Отладка⇒Отладчик) в главном окне и затем запустить сценарий, выбрав пункт меню Run⇒Run Module в окне текстового редактора. После активизации отладчика в коде можно устанавливать точки останова, которые вызывают паузу в выполнении, щелкнув правой кнопкой мыши на строках в окнах текстового редактора, просматривать значения переменных и т.д. Во время отладки можно также следить за выполнением программы – по мере прохождения по коду будет отображаться строка, выполняемая в текущий момент.

Для более простых операций отладки можно щелкнуть правой кнопкой мыши на тексте сообщения об ошибке, быстро переходя на строку кода, где возникла ошибка, исправлять ее и запускать заново. Вдобавок текстовый редактор IDLE предлагает крупную коллекцию дружественных к программисту инструментов, в числе которых не рассматриваемые здесь расширенные операции поиска. Поскольку в IDLE применяются интуитивно понятные взаимодействия с графическим пользовательским интерфейсом, вы должны поэкспериментировать с системой вживую, чтобы получить представление об остальных инструментах.

Замечания по использованию: IDLE

Среда IDLE бесплатна, проста в применении, переносима и автоматически доступна на большинстве платформ. Я обычно рекомендую ее новичкам в Python, т.к. она облегчает ряд деталей запуска и не требует опыта работы с командной строкой системы. Однако возможности IDLE несколько ограничены в сравнении с более развитыми коммерческими IDE-средами, а некоторым использование IDLE может показаться более трудным, чем набор команд в командной строке. Ниже перечислены моменты, о которых должны помнить начинающие пользователи IDLE, чтобы избежать распространенных ловушек.

- Вы обязаны явно добавлять расширение .ru к именам файлов. Об этом уже упоминалось при обсуждении файлов, но оно является общим камнем преткновения в IDLE, особенно для пользователей Windows. При сохранении файлов IDLE не добавляет расширение .ru к их именам автоматически. Сохраняя файл в первый раз, самостоятельно набирайте расширение .ru. В противном случае, несмотря на возможность запуска файла из IDLE (и командной строки), вы не сможете импортировать его ни в интерактивной подсказке, ни в других модулях.

- Выполняйте сценарии, выбирая пункт меню Run⇒Run Module в окнах текстового редактора, а не посредством интерактивного импортирования и перезагрузки. Ранее в главе было показано, что выполнить файл допускается за счет его интерактивного импортирования. Тем не менее, такая схема может усложниться, потому что она требует ручной перезагрузки файлов после внесения в них изменений. По контрасту с этим применение пункта меню Run⇒Run Module в IDLE всегда приводит к выполнению текущей версии файла, что похоже на его запуск в командной строке системы. При необходимости среда IDLE также предлагает сначала сохранить файл (еще одна распространенная ошибка за рамками IDLE).
- Вам нужно перезагружать только модули, подвергающиеся интерактивному тестированию. Подобно командной строке системы пункт меню Run⇒Run Module в IDLE всегда обеспечивает выполнение текущей версии файла верхнего уровня и любых импортируемых им модулей. По этой причине Run⇒Run Module устраняет часто возникающие недоразумения, связанные с импортированием. Вы должны перезагружать только модули, которые интерактивно импортируете и тестируете в IDLE. Если вместо пункта меню Run⇒Run Module вы решили использовать прием с импортированием и перезагрузкой, то не забывайте о возможности перехода к ранее вводимым командам с помощью клавиатурных комбинаций <Alt+P>/<Alt+N>.
- Вы можете настраивать IDLE. Чтобы изменить шрифты и цвет отображения текста в IDLE, выберите пункт Configure (Настройка) в меню Options (Параметры) любого окна IDLE. Вы можете также конфигурировать действия клавиатурных комбинаций, настройки отступов, автоматическое завершение и многое другое; дополнительные сведения ищите в справочной системе IDLE.
- В настоящее время в IDLE отсутствует возможность очистки экрана. Похоже, что это частый запрос (вероятно потому, что такая возможность доступна в похожих IDE-средах), который когда-нибудь будет решен. Однако на сегодняшний день нет никакого способа очистки интерактивного окна от имеющегося текста. Если вы хотите, чтобы текст в окне исчез, тогда можете либо нажать и удерживать клавишу <Enter>, либо набрать цикл Python для вывода последовательности пустых строк (конечно, мало кто будет применять второй прием, но он выглядит более высокотехнологичным, чем нажатие клавиши <Enter>!).
- Программы с графическим пользовательским интерфейсом на основе tkinter и многопоточные программы могут не работать нормально с IDLE. Поскольку IDLE является программой Python/tkinter, она может зависнуть при выполнении в ней некоторых видов сложных программ Python/tkinter. В более новых версиях IDLE, которые запускают пользовательский код в одном процессе, а сам графический интерфейс IDLE в другом, проблем с этим стало меньше, но определенные программы (особенно использующие многопоточную обработку) могут по-прежнему приводить к зависанию графического интерфейса. Даже вызова в коде функции quit из tkinter (обычного способа завершения программы с графическим пользовательским интерфейсом) может оказаться достаточно для зависания графического интерфейса вашей программы при ее выполнении в IDLE (здесь может быть лучше только destroy). Ваш код может не страдать от проблем такого рода, но запомните в качестве эмпирического правила: применять IDLE для редактирования программ с графическим пользовательским интерфейсом всегда безопасно, но запускать их лучше другими способами, такими как использование щелчков на значках или окна командной строки. Если вы

сомневаетесь, будет ли ваш код нормально работать в IDLE, то попробуйте его запустить вне этой среды.

- В случае возникновения ошибок подключения попробуйте запустить IDLE в режиме с единственным процессом. Похоже, данная проблема в последних версиях Python исчезла, но с ней могут столкнуться те читатели, кто имеет дело с более старыми версиями. Из-за того, что IDLE требует взаимодействий между отдельными процессами пользователя и графического пользовательского интерфейса, временами могут возникать затруднения с запуском на определенных платформах (в частности, иногда IDLE отказывается запускаться на компьютерах Windows по причине блокирования подключений со стороны брандмауэра). Если произошла ошибка подключения, то всегда можно запустить среду IDLE с помощью командной строки, которая заставит ее выполнятся в режиме с единственным процессом без подпроцесса для пользовательского кода, что устранит проблемы с взаимодействием: такой режим активизируется флагом командной строки `-n`. Скажем, на компьютере Windows откройте окно командной строки и введите команду `idle.py -n`, находясь в каталоге `C:\Python33\Lib\idlelib` (при необходимости сначала выполнив команду `cd`). Команда `python -m idlelib.idle -n` работает откуда угодно (флаг `-m` описан в приложении А второго тома).
- Проявляйте осторожность в отношении некоторых удобных функциональных возможностей IDLE. Среда IDLE делает многое, чтобы облегчить жизнь новичкам, но некоторые ее трюки неприменимы за пределами графического пользовательского интерфейса IDLE. Например, IDLE запускает ваши сценарии в собственном интерактивном пространстве имен, поэтому переменные из вашего кода автоматически видны в интерактивном сеансе IDLE – вам не придется постоянно выполнять команды `import` для доступа к именам на верхнем уровне уже выполненных файлов. Это удобно, но может стать источником путаницы, поскольку за рамками среды IDLE имена всегда должны явно импортироваться из файлов, чтобы их можно было использовать.

Когда вы запускаете файл кода, IDLE также автоматически переходит в каталог с этим файлом и добавляет данный каталог в путь поиска модулей при импортировании – удобная возможность, которая позволяет задействовать файлы и импортировать модули без настроек пути поиска, но не доступна при запуске файлов вне IDLE. Применять такие средства вполне normally, но не забывайте, что они относятся к поведению IDLE, а не Python.

Другие IDE-среды

Поскольку IDLE является бесплатной, переносимой и стандартной частью Python, она будет хорошим первым инструментом для разработки, с которым следует ознакомиться, чтобы понять, собираетесь ли вы вообще использовать IDE-среду. Если вы только начинаете изучение Python, то я рекомендую применять IDLE для проработки упражнений, приводимых в книге, при условии, что вы не знакомы и не отдаете предпочтение режиму разработки на основе командной строки. Тем не менее, существует несколько альтернативных IDE-сред для разработчиков на Python, ряд которых значительно мощнее и надежнее, чем IDLE.

Ниже кратко описаны самые часто используемые IDE-среды для Python.

Eclipse и PyDev

Eclipse – развитая IDE-среда с графическим пользовательским интерфейсом и открытым кодом. Первоначально разработанная как IDE-среда для Java, Eclipse также поддерживает разработку на Python в случае установки подключаемого модуля PyDev (или его аналога). Среда Eclipse представляет собой популярный и мощный вариант для разработки на Python и выходит далеко за рамки набора функциональных возможностей IDLE. Она включает поддержку завершения кода, цветовую раскраску синтаксиса, синтаксический анализ, рефакторинг, отладку и т.д. Недостатки заключаются в том, что это крупная система в плане установки, и для ряда функциональных возможностей она требует условно-бесплатных расширений (со временем ситуация может измениться). Однако когда вы закончите с IDLE, то сочетание Eclipse/PyDev заслуживает вашего внимания.

Komodo

Будучи полнофункциональной средой с графическим пользовательским интерфейсом для разработки на Python (и на других языках), Komodo поддерживает стандартную цветовую раскраску синтаксиса, редактирование текста, отладку и другие возможности. В добавок Komodo предлагает много расширенных средств, отсутствующих в IDLE, включая файлы проектов, интеграцию с системой управления версиями и отладку регулярных выражений. На момент написания книги среда Komodo не была бесплатной, но просматривайте ее текущее состояние на веб-сайте компании ActiveState (<http://www.activestate.com>), которая также предлагает дистрибутивный пакет ActivePython, упоминаемый в приложении А второго тома.

NetBeans IDE для Python

NetBeans – это мощная среда с графическим пользовательским интерфейсом и открытым кодом, обладающая поддержкой многочисленных расширенных возможностей для разработчиков на Python: завершение кода, автоматическое добавление отступов и цветовая раскраска синтаксиса, подсказки редактора, свертывание кода, рефакторинг, отладка, покрытие кода и тестирование, проекты и многое другое. Она может применяться для разработки кода CPython и Jython. Подобно Eclipse среда NetBeans требует большего числа шагов при установке, чем IDLE, но многие считают, что она вполне стоит приложенных усилий. Поищите в веб-сети последнюю информацию и ссылки.

PythonWin

PythonWin – бесплатная IDE-среда для Python, функционирующая только под управлением Windows, которая поставляется как часть дистрибутива ActivePython от ActiveState (и также может быть загружена отдельно из ресурсов <http://www.python.org>). Она примерно похожа на IDLE с добавлением нескольких полезных расширений, специфичных для Windows; скажем, PythonWin имеет поддержку COM-объектов. В наши дни IDLE вероятно более эффективна, чем PythonWin (например, архитектура с двумя процессами IDLE часто предотвращает зависание). Тем не менее, PythonWin по-прежнему предлагает инструменты для разработчиков под Windows, которые в IDLE отсутствуют. Дополнительные сведения ищите на веб-сайте <http://www.activestate.com>.

Wing, Visual Studio и прочие

Среди разработчиков на Python популярны и другие IDE-среды, в том числе главным образом коммерческая *Wing IDE*, Microsoft *Visual Studio* через подключаемый модуль, а также *PyCharm*, *PyScripter*, *Pyshield* и *Spyder*, но в книге нет места для их полноценного описания, и со временем, несомненно, появятся новые среды. Фактически теперь почти каждый дружественный к программистам *текстовый редактор* имеет какую-то разновидность поддержки для разработки на Python, будь она устанавливаемой сразу или получаемой отдельно. Скажем, *Emacs* и *Vim* располагают значительной поддержкой Python.

Выбор IDE-среды зачастую субъективен, поэтому я рекомендую отыскать инструменты, которые согласуются с вашим стилем и целями разработки. Больше сведений можно получить, просматривая ресурсы на веб-сайте <http://www.python.org> или производя поиск по ключевым словам “Python IDE”. По ссылке <https://wiki.python.org/moin/PythonEditors> доступна информация о десятках IDE-сред и текстовых редакторов, пригодных для программирования на Python.

Другие варианты запуска

К этому моменту вы узнали, как выполнять код, набранный интерактивно, и каким образом запускать код, сохраненный в файлах — с помощью командной строки, щелчков на значках, `import` и `exec`, графических пользовательских интерфейсов вроде IDLE и т.д. Было охвачено большинство обычно используемых приемов, которых вполне достаточно для запуска кода, приводимого в книге. Однако существуют дополнительные способы выполнения кода Python, большинство из которых играют специальные или узкие роли. Ради полноты некоторые из них кратко описаны в последующих разделах.

Встраивание вызовов

В ряде специализированных областей код Python может запускаться автоматически включающей его системой. В подобных случаях мы говорим, что программы Python являются *встроенным* в другую программу (т.е. запускаются ею). Сам код Python может находиться в текстовом файле, храниться в базе данных, извлекаться из HTML-страницы, получаться в результате разбора XML-документа и т.д. Но с практической точки зрения не вы, а другая система сообщает Python о необходимости запуска созданного вами кода.

Такой режим встроенного выполнения обычно применяется для поддержки настройки у конечного пользователя — например, игровая программа может допускать изменения в игре за счет запуска в стратегически важные моменты времени встроенного кода Python, доступного пользователям. Пользователи имеют возможность модифицировать систему такого типа, предоставляя или изменяя код Python. Поскольку код Python интерпретируется, отсутствует потребность в перекомпиляции всей системы для внедрения изменений (выполнение кода Python обсуждалось в главе 2).

В этом режиме включающая система, которая запускает ваш код, может быть написана на C, C++ или даже Java в случае использования Jython. Скажем, строки кода Python вполне возможно создавать и запускать из программы C, вызывая функции API-интерфейса Python (набор служб, экспортруемых библиотеками, которые создаются при компиляции Python на компьютере):

```
#include <Python.h>
...
Py_Initialize(); // Это C, а не Python
PyRun_SimpleString("x = 'brave ' + 'sir robin'"); // Но данный вызов
// выполняет код Python
```

В приведенном фрагменте кода С интерпретатор Python внедряется путем связывания его библиотек и затем ему передается на выполнение строка с оператором присваивания Python. Программы С способны получать доступ к модулям и объектам Python, а также обрабатывать или выполнять их с применением других инструментов API-интерфейса Python.

Хотя книга не об интеграции Python/C, вы должны знать, что в зависимости от того, каким образом ваша организация планирует использовать Python, вы можете быть или не быть тем, кто действительно запускает создаваемые вами программы Python. Невзирая на это, обычно вы по-прежнему можете применять интерактивный и основанный на файлах приемы, чтобы тестировать код в изоляции от включающей системы, которая будет в итоге его использовать⁵.

Фиксированные двоичные исполняемые файлы

Фиксированные двоичные исполняемые файлы, описанные в главе 2, представляют собой пакеты, которые объединяют байт-код вашей программы и интерпретатор Python в единую исполняемую программу. Такой подход позволяет запускать программы Python теми же способами, которыми запускалась бы любая другая исполняемая программа (щелчок на значке, командная строка и т.п.). Несмотря на то что этот прием удобен для доставки программных продуктов, он не предназначен для применения во время их разработки; фиксация обычно делается прямо перед доставкой (после того, как разработка окончена). Дополнительные сведения ищите в предыдущей главе.

Варианты запуска из текстовых редакторов

Как упоминалось ранее, не являясь полнофункциональными IDE-средами с графическим пользовательским интерфейсом, большинство дружественных к программисту текстовых редакторов располагают поддержкой для редактирования и возможно запуска программ Python. Такая поддержка может быть встроенной или загружаемой из веб-сети. Например, если вы знакомы с текстовым редактором Emacs, тогда можете редактировать и запускать весь свой код Python изнутри этого текстового редактора. Дополнительные сведения доступны по ссылке <https://www.python.org/doc/editors>.

Прочие варианты запуска

В зависимости от платформы могут существовать дополнительные способы запуска программ Python. Скажем, на некоторых системах Macintosh у вас может быть возможность перетаскивать значки файлов с программами Python на значок интерпретатора Python, чтобы выполнить их. На системах Windows вы всегда можете запускать сценарии Python с помощью пункта Выполнить меню Пуск. Вдобавок стандартная библиоте-

⁵ Больше сведений о встраивании Python в C/C++ можно почерпнуть из книги *Programming Python* (<http://shop.oreilly.com/product/9780596158118.do>). API-интерфейс внедрения может напрямую вызывать функции Python, загружать модули и т.д. Также имейте в виду, что система Jython позволяет программам Java вызывать код Python с применением API-интерфейса, основанного на Java (класса интерпретатора Python).

ка Python имеет утилиты, которые позволяют программам Python запускаться другими программами Python в отдельных процессах (например, `os.popen`, `os.system`), а сценарии Python могут также порождать процессы в более крупных контекстах вроде веб-сети (например, веб-страница способна вызвать сценарий на сервере). Тем не менее, варианты подобного рода выходят за рамки материала настоящей главы.

Будущие возможности?

В главе отражается текущая практика, но большая часть материала специфична для платформ и времени. Действительно, многие представленные здесь детали выполнения и запуска появились на протяжении периода, в течение которого существовали различные издания данной книги. Как и с вариантами выполнения программ, не исключено, что со временем могут возникнуть новые варианты запуска программ.

Новые операционные системы и новые версии существующих систем также могут предложить методы выполнения, не входящие в число рассмотренных в главе. В целом из-за того, что Python идет в ногу с такими изменениями, вы должны быть в состоянии запускать программы Python любым способом, который подходит к используемым вами компьютерам, как сейчас, так и в будущем — будь то постукивание экрана планшета или смартфона, захватывание значков в виртуальной реальности или произношение вслух имени сценария в разговоре с коллегами.

Изменения реализации также могут как-то повлиять на схемы запуска (например, полный компилятор производил бы обычные исполняемые файлы, которые запускаются во многом подобно фиксированным двоичным файлам в наши дни). Если бы я знал, что сулит нам будущее, то вероятно общался бы с биржевым маклером, а не писал эти слова!

Какой вариант должен использоваться?

При таком обилии вариантов у настоящего новичка может возникнуть вполне естественный вопрос: какой вариант будет наилучшим для меня? Если вы только приступили к освоению Python, то в общем случае должны опробовать продукт IDLE. Он предоставляет дружественную к пользователю среду с графическим интерфейсом и скрывает ряд внутренних деталей конфигурации. Он также поступает с нейтральным к платформам текстовым редактором для написания кода сценариев и является стандартной бесплатной частью системы Python.

С другой стороны, если вы — опытный программист, то вам может быть более удобно в одном окне работать с выбранным текстовым редактором, а в другом запускать редактируемые программы посредством командной строки и щелчков на значках. (На самом деле именно так я пишу программы Python, но это объясняется использованием Unix в далеком прошлом.) Поскольку выбор сред для разработки крайне субъективен, в плане универсальных рекомендаций ничего больше я предложить не могу. В общем, любая среда, в которой вам нравится работать, и будет для вас наилучшей.

Отладка кода Python

Естественно, никто из моих читателей и студентов никогда не допускал ошибки в своем коде (*здесь должен быть смайлик*). Но для менее удачливых ваших друзей ниже представлен краткий обзор стратегий, которые часто применяются реальными программистами на Python для отладки кода, чтобы вы могли к ним обращаться, когда всерьез займетесь написанием кода.

- **Ничего не делайте.** Под этим вовсе не понимается то, что программисты на Python не отлаживают свой код, но когда вы допускаете ошибку в программе Python, то получаете полезное и удобочитаемое сообщение об ошибке (вскоре вы увидите их, если только уже не видели). Если вы хорошо знаете Python, особенно в случае собственного кода, тогда часто этого достаточно – прочтите сообщение об ошибке и исправьте отмеченную в нем строку файла. Для многих такой подход и является отладкой в Python. Однако он не всегда будет идеальным в случае более крупных систем.
- **Вставьте операторы `print`.** Вероятно основной способ, которым программисты на Python отлаживают свой код (и способ, которым пользуюсь я сам), предусматривает вставку операторов `print` и повторное выполнение кода. Из-за того, что код Python запускается немедленно после внесения изменений, обычно он будет самым быстрым методом получения большего объема информации, чем предоставляет сообщение об ошибке. Операторы `print` не обязаны быть сложно устроеными – простой фразы “Я здесь” или вывода значений переменных, как правило, достаточно для снабжения необходимым контекстом. Нужно лишь не забыть об удалении либо комментировании (т.е. помещении перед ними `#`) отладочных операторов `print` до поставки кода!
- **Применяйте отладчики IDE-сред с графическим пользовательским интерфейсом.** В случае крупных систем, которые писали не вы, и начинающих, кого интересует более детальная трассировка кода, большинство IDE-сред для разработки на Python располагают определенной поддержкой отладки. Среда IDLE также содержит отладчик, но на практике он не слишком часто используется – то ли из-за того, что он не имеет командной строки, то ли потому, что добавить операторы `print` обычно быстрее, чем настроить сеанс отладки с графическим пользовательским интерфейсом. Дополнительные сведения поищите в справочной системе IDLE или просто поэкспериментируйте; базовый интерфейс отладчика описан в разделе “Расширенные инструменты IDLE” ранее в главе. Другие IDE-среды, такие как Eclipse, NetBeans, Komodo и Wing IDE, предлагают более развитые отладчики; если вы работаете с ними, то обратитесь в документацию.
- **Применяйте отладчик командной строки `pdb`.** Для максимального контроля в состав Python входит отладчик исходного кода по имени `pdb`, доступный как модуль в стандартной библиотеке Python. В отладчике `pdb` вы набираете команды для выполнения кода строка за строкой, отображения переменных, добавления и удаления точек останова, продолжения до точки останова или до возникновения ошибки и т.д. Вы можете запустить `pdb` интерактивно или путем импортирования либо как сценарий верхнего уровня. В любом случае, поскольку вы можете набирать команды для управления сеансом, он представляет собой мощный инструмент отладки. Отладчик `pdb` также располагает функцией анализа причин произошедшего (`pdb.pm()`), которую можно выполнять после того, как возникло исключение, чтобы получить информацию в момент появления ошибки. Более подробные сведения `pdb` предлагаются в руководстве по библиотеке Python и в главе 36 второго тома, а в приложении А второго тома приводится пример запуска `pdb` как сценария с помощью аргумента `-m` команды `python`.
- **Применяйте аргумент `-i` команды `python`.** Даже не добавляя операторы `print` и не запуская под управлением `pdb`, при возникновении ошибок вы все равно можете выяснить, что пошло не так. Если вы запустите свой сценарий в командной строке, указав аргумент `-i` (например, `python -i m.py`), то Python войдет в режим интерактивного интерпретатора (с подсказкой `>>>`), когда сценарий закончит работу, либо успешно дойдя до конца, либо из-за возникновения ошибки. В этот момент вы можете вывести финальные значения переменных, чтобы получить больше деталей о том, что случилось в коде, т.к. они находятся в пространс-

тве имен верхнего уровня. Вы можете затем импортировать и запустить отладчик pdb для даже более точного определения контекста; его режим анализа причин произошедшего позволит изучить последнюю ошибку, если сценарий потерпел неудачу. В приложении А второго тома демонстрируется аргумент -i в действии.

- **Другие возможности.** В случае более специфических требований к отладке, таких как поддержка многопоточных программ, встроенный код и присоединение к процессам, вы можете поискать дополнительные инструменты в области проектов с открытым кодом. Скажем, система Winpdb является автономным отладчиком с расширенной поддержкой отладки, оснащенная межплатформенным графическим пользовательским и консольным интерфейсом.

Перечисленные варианты станут более важными, когда мы начнем писать крупные сценарии. Тем не менее, вероятно лучшие новости в плане отладки касаются того, что в Python ошибки обнаруживаются и протоколируются, а не тихо пропускаются или вызывают полный отказ системы. Фактически сами ошибки представляют собой четко определенный механизм, известный как *исключения*, которые можно перехватывать и обрабатывать (исключения подробно рассматриваются в части VII). Конечно, допускать ошибки никогда не было приятным занятием, но спросите у тех, кто еще помнит времена, когда отладка означала сосредоточенное изучение кипы распечаток содержимого памяти с шестнадцатеричным калькулятором в руках: поддержка отладки в Python делает ошибки гораздо менее болезненными, нежели они были бы иначе.

Резюме

В главе мы взглянули на распространенные способы запуска программ Python: выполнение кода, набранного интерактивно, и выполнение кода, сохраненного в файлах, с помощью командной строки системы, щелчков на значках файлов, импортирования модулей, вызовов exec и IDE-сред с графическим пользовательским интерфейсом, таких как IDLE. Мы раскрыли здесь многочисленные практические аспекты запуска. Цель главы заключалась в том, чтобы снабдить вас информацией, достаточной для самостоятельного написания кода, что и будет делаться в следующей части книги. Там мы приступим к исследованию самого языка Python, начиная с его основных *типов данных* – объектов, которые являются субъектами ваших программ.

Но для начала ответьте на традиционные контрольные вопросы, чтобы попрактиковаться с тем, что вы изучили в главе. Поскольку это последняя глава части, далее предлагаются набор более полных упражнений, которые проверят ваши знания тематики целой части. Для получения помощи в решении упражнений или просто ради повторения просмотрите приложение после того, как попробуете самостоятельно решить упражнения.

Проверьте свои знания: контрольные вопросы

1. Как можно запустить сеанс интерактивного интерпретатора?
2. Где набирается команда строка для запуска файла сценария?
3. Назовите четыре или больше способов для запуска кода, хранящегося в файле сценария.
4. Назовите две ловушки, связанные со щелчками на значках файлов в Windows.

5. Почему может потребоваться перезагрузка модуля?
6. Как запустить сценарий из IDLE?
7. Назовите две ловушки, связанные с использованием IDLE.
8. Что такое пространство имен и как оно связано с файлами модулей?

Проверьте свои знания: ответы

1. Чтобы запустить интерактивный сеанс в Windows 7 и предшествующих версиях, необходимо щелкнуть на кнопке Пуск, выбрать пункт Все программы, щелкнуть на элементе Python для нужной версии и выбрать пункт Python (command line) (Python (командная строка)). Добиться того же самого результата в Windows и на других платформах можно путем набора командной строки `python` в консольном окне системы (окне командной строки в Windows). Еще одной альтернативой будет запуск среды IDLE, т.к. главное окно оболочки Python является интерактивным сеансом. В зависимости от платформы и версии Python, если переменная среды PATH не была установлена для нахождения Python, то может понадобиться переход посредством команды `cd` в каталог, где установлен Python, или указание полного пути к каталогу, а не просто `python` (например, `C:\Python33\python` в Windows при условии, что запускающий модуль Python 3.3 не применяется).
2. Вы набираете командные строки внутри того, что ваша платформа предлагает в качестве системной консоли: окне командной строки в Windows, окне `xterm` или терминала в Unix, Linux и Mac OS X и т.д. Вы набираете их в строке с приглашением системы, а не приглашением интерактивного интерпретатора Python (`>>>`) – не перепутайте их.
3. Код в файле сценария (в действительности модуля) можно запускать с помощью командной строки системы, щелчков на значках файлов, импортирования и перезагрузки, встроенной функции `exec`, а также меню выбранной IDE-среды с графическим пользовательским интерфейсом наподобие пункта меню `Run⇒Run Module` в IDLE. В Unix их также можно запускать как исполняемые файлы посредством трюка с `#!`, а некоторые платформы поддерживают более специализированные методики запуска (например, перетаскивание). Вдобавок ряд текстовых редакторов обладает уникальными возможностями запуска кода Python, определенные программы Python предоставляются в виде автономных фиксированных двоичных исполняемых файлов, а некоторые системы используют код Python во встроенном режиме, запуская его включающей программой, которая написана на языке C, C++ или Java. Последний прием обычно преследует цель снабжения пользователей уровнем настройки.
4. Сценарии, которые что-то выводят и затем заканчивают работу, приводят к немедленному исчезновению окна вывода, прежде чем вы сможете его разглядеть (именно тогда трюк с `input` оказывается полезным); сообщения об ошибках, генерируемые сценарием, также появляются в окне вывода, которое закрывается до того, как вы успеете изучить содержимое сообщений (одна из причин того, что для большей части разработки лучше применять командную строку системы и IDE-среды вроде IDLE).

5. По умолчанию Python импортирует (загружает) модуль только один раз на процесс, поэтому если вы изменили его исходный код и хотите выполнить новую версию, не останавливая и не перезапуская Python, то должны перезагрузить модуль. Прежде чем модуль можно будет перезагрузить, его потребуется хотя бы раз импортировать. Запуск файлов кода в командной строке системы, посредством щелчков на значках или через IDE-среду, подобную IDLE, в целом делает это ненужным, потому что такие схемы запуска обычно каждый раз выполняют текущую версию файла исходного кода.
6. Находясь внутри окна текстового редактора с файлом, который необходимо запустить, выберите пункт меню Run⇒Run Module. Это приведет к запуску исходного кода в окне как файла сценария верхнего уровня и отображению его вывода в окне интерактивной оболочки Python.
7. Среда IDLE по-прежнему может зависать при запуске программ определенных типов — особенно программ с графическим пользовательским интерфейсом, в которых применяется многопоточная обработка (расширенная методика, выходящая за рамки настоящей книги). Кроме того, в IDLE имеется несколько удобных функциональных возможностей, которые способны сбивать с толку, когда вы покинете IDLE: например, переменные сценария автоматически импортируются в интерактивную оболочку IDLE, а рабочие каталоги изменяются при запуске файла, но сам Python такие действия не предпринимает.
8. Пространство имен — это всего лишь пакет переменных (имен). В Python он принимает форму объекта с атрибутами. Каждый файл модуля автоматически является пространством имен, т.е. пакетом переменных, которые отражают присваивания, сделанные на верхнем уровне файла. Пространства имен помогают избегать конфликтов имен в программах Python: поскольку каждый файл модуля представляет собой самостоятельное пространство имен, файлы должны явно импортировать другие файлы, чтобы использовать их имена.

Проверьте свои знания: упражнения для части I

Наступило время для самостоятельного написания кода. Первое собрание упражнений будет довольно простым, но оно предназначено для того, чтобы удостовериться в том, что вы готовы к проработке оставшихся материалов книги, а несколько вопросов в них намекают на темы в последующих главах. Сверьтесь с ответами в приложении; упражнения и их решения временами содержат дополнительную информацию, не обсуждаемую в основном тексте, поэтому вы должны взглянуть на решения, даже если собираетесь самостоятельно отвечать на все вопросы.

- 1. Взаимодействие.** Используя командную строку системы, IDLE или любой другой метод, который работает на вашей платформе, запустите интерактивную оболочку Python (приглашение `>>>`) и наберите выражение "Hello World!" (включая кавычки). Стока должна отобразиться повторно. Цель упражнения — убедиться, что ваша среда сконфигурирована для запуска Python. В некоторых сценариях может потребоваться сначала выполнить команду `cd`, набрать полный путь к исполняемому файлу Python или добавить данный путь к переменной среды PATH. При желании можете установить PATH в файле `.cshrc` или `.kshrc`, чтобы сделать Python постоянно доступным в системах Unix; в Windows это обычно делается с применением графического пользовательского интерфейса для установки переменных среды. Дополнительные сведения ищите в приложении А (том 2).

- 2. Программы.** С помощью выбранного текстового редактора напишите простой файл модуля, содержащий единственный оператор `print('Hello module world!')`, и сохраните его как `module1.py`. Запустите файл, используя любой желаемый вариант: запуск в IDLE, щелчок на значке файла, передача его интерпретатору Python в командной строке системы (например, `python module1.py`), вызов встроенной функции `exec`, импортирование и перезагрузка и т.д. В сущности, поэкспериментируйте с максимально возможными приемами запуска файла, которые обсуждались в книге. Какая методика выглядит самой легкой? (Разумеется, единственno правильного ответа здесь нет.)
- 3. Модули.** Запустите интерактивную командную строку Python (приглашение `>>>`) и импортируйте модуль, написанный в упражнении 2. Попробуйте переместить файл в другой каталог и снова импортируйте его из первоначального каталога (т.е. для импортирования запустите Python в первоначальном каталоге). Что произошло? (Подсказка: присутствует ли файл байт-кода `module1.pyc` в первоначальном каталоге или что-то подобное в подкаталоге `__pycache__`?)
- 4. Сценарии.** Если ваша платформа поддерживает это, тогда добавьте строку `# !` в начало файла модуля `module1.py`, назначьте ему права на выполнение и запустите его напрямую как исполняемый файл. Что должна содержать первая строка? Стока `# !` обычно имеет смысл только на платформах Unix, Linux и Unix-подобных вроде Mac OS X. Если вы работаете в Windows, тогда взамен попробуйте запустить свой файл, указав в окне командной строки только его имя без слова `python` перед ним (что актуально для последних версий Windows), используя диалоговое окно, которое доступно через пункт меню `Start⇒Run...`, или еще как-нибудь. Если вы применяете версию Python 3.3 или запускающий модуль для Windows, который устанавливается вместе с ней, то поэкспериментируйте с изменением строки `# !` сценария для запуска разных версий Python, которые могут быть установлены на компьютере (или проработайте руководство в приложении Б второго тома).
- 5. Ошибки и отладка.** Поэкспериментируйте с набором математических выражений и присваиваний в интерактивной командной строке Python. Попутно наберите выражения `2 ** 500` и `1 / 0` и сошлитесь на неопределенное имя переменной, как делалось ранее в главе. Что произошло?
- Возможно, вам пока неизвестно, что когда вы допускаете ошибку, то предпринимаете обработку исключений: тема, подробно исследуемая в части VII. Как там будет показано, формально вы инициируете то, что называется *стандартным обработчиком исключений* – логика, которая выводит стандартное сообщение об ошибке. Если вы не перехватываете ошибку, тогда ее перехватит стандартный обработчик и выведет в ответ соответствующее сообщение об ошибке.
- Исключения также связаны с понятием *отладки* в Python. Когда вы только начинаете, стандартное сообщение об ошибке Python, отображаемое в случае исключения, вероятно, обеспечит всю необходимую поддержку обработки исключений – оно указывает причину ошибки и строку в коде, которая была активной при возникновении ошибки. Дополнительные сведения об отладке даны во врезке “Отладка кода Python” ранее в главе.
- 6. Прерывание работы и циклы.** В командной строке Python наберите следующий код:

```
L = [1, 2]          # Создать список из двух элементов
L.append(L)        # Добавить L как одиничный элемент к самому себе
L                  # Вывести L: циклический/закольцованный объект
```

Что произойдет? Во всех недавних версиях Python вы увидите странный вывод, который будет описан в приложении с решениями и станет более понятным, когда начнется изучение языка в следующей части книги. Если вы используете версию Python старее, чем 1.5.1, тогда нажатие клавиатурной комбинации <Ctrl+C>, скорее всего, поможет на большинстве платформ. Как вы думаете, почему ваша версия Python реагирует именно так на этот код?



Если вы имеете дело с версией, предшествующей Python 1.5.1 (надо надеяться, что это редкий сценарий в наши дни!), тогда перед запуском данного теста удостоверьтесь в том, что ваш компьютер способен остановить программу с помощью клавиатурной комбинации <Ctrl+C>, или же вам придется ожидать длительное время.

7. Документация. Прежде чем продолжить, потратьте хотя бы 15 минут на просмотр руководств по библиотеке и языку Python, чтобы получить представление о доступных инструментах в стандартной библиотеке и структуре комплекта документации. За указанное минимальное время вы ознакомитесь с местоположением основных тем в руководствах; после этого вам будет легко находить то, что нужно. Руководства доступны через программную группу Python в Windows, через пункт Python Docs (Документация по Python) меню Help (Справка) в IDLE или по ссылке <http://www.python.org/doc>. В главе 15 будут даны дополнительные сведения о руководствах и других источниках документации (включая PyDoc и функцию help). При наличии свободного времени исследуйте веб-сайт Python, а также хранилище сторонних расширений PyPI. Особенное внимание уделите документации и поисковым возможностям Python.org (<http://www.python.org>); они могут служить важными ресурсами.

ЧАСТЬ II

Типы и операции

Введение в типы объектов Python

В настоящей главе начинается наше путешествие по языку Python. Неформально в Python мы *делаем дела с помощью оснащения*¹. “Дела” принимают форму операций, подобных сложению и конкатенации, а “оснащение” относится к объектам, на которых мы выполняем такие операции. В этой части книги внимание будет сосредоточено на *оснащении* и на *делах*, которые наши программы могут делать с его помощью.

Выражаясь чуть более формально, данные в Python имеют форму *объектов* – либо встроенных объектов, предоставляемых Python, либо объектов, которые мы создаем с применением классов Python или внешних языковых инструментов, таких как библиотеки расширений C. Хотя позже мы уточним приведенное определение, объекты по существу представляют собой порции памяти со значениями и наборами связанных операций. Как вы увидите, в сценарии Python абсолютно *все* является объектами. Даже простые числа определяются значениями (например, 99) и поддерживаемыми операциями (сложение, вычитание и т.д.).

Поскольку объекты – наиболее фундаментальное понятие в программировании на Python, мы начнем главу с обзора встроенных типов объектов Python. В последующих главах мы охватим недостающие в обзоре детали. Здесь нашей целью является краткое введение в основы.

Концептуальная иерархия Python

Прежде чем переходить к коду, давайте первым делом проясним, как эта глава вписывается в общую картину Python. В действительности программы Python могут быть разложены на модули, операторы, выражения и объекты, как описано ниже.

1. Программы состоят из модулей.
2. Модули содержат операторы.
3. Операторы содержат выражения.
4. *Выражения создают и обрабатывают объекты.*

При обсуждении модулей в главе 3 был представлен наивысший уровень такой иерархии. В главах этой части мы начинаем снизу – исследуем встроенные объекты и выражения, которые можно писать для их использования.

¹ Простите за формальность. Я специалист в области компьютерных наук.

Мы перейдем к изучению операторов в следующей части книги, хотя обнаружим, что они в значительной степени существуют для управления объектов, которые будут встречаться в данной части. Кроме того, к тому времени, как мы доберемся до классов в части, посвященной объектно-ориентированному программированию, мы выясним, что они дают нам возможность определять собственные новые типы объектов за счет применения и моделирования типов объектов, которые будут исследоваться здесь. Учитывая все сказанное, встроенные объекты являются обязательной отправной точкой для всего путешествия по Python.



В традиционных вводных курсах по программированию часто делается акцент на трех главных принципах: очередность (“Делай это, затем то”), выбор (“Делай это, если то истинно”) и повторение (“Делай это много раз”). В Python есть инструменты всех трех категорий наряду с инструментами для определения функций и классов. Упомянутые темы могут помочь упорядочить мысли на раннем этапе, но они имеют несколько искусственный и упрощенческий характер. Например, выражения вроде списковых включений представляют собой и повторение, и выбор; некоторые из этих терминов в Python имеют другой смысл, а многие вводимые позже концепции, кажется, вообще не вписываются в указанный шаблон. В Python более сильным объединяющим принципом являются объекты и то, что мы можем с ними делать. Чтобы понять почему, читайте дальше.

Для чего используются встроенные типы?

Если вы имели дело с языком более низкого уровня наподобие C или C++, то знаете, что большая часть работы сконцентрирована на реализации *объектов*, также известных, как *структуры данных*, для представления компонентов в вашей предметной области. Вам нужно планировать структуры в памяти, управлять выделением памяти, создавать процедуры поиска и доступа и т.д. Такие рутинные задачи утомительны (и чреваты ошибками) и они зачастую отвлекают от истинных целей программы.

В типичных программах большинство рутинных работ исчезает. Из-за того, что Python предоставляет мощные типы объектов как неотъемлемую часть языка, обычно нет необходимости в написании кода реализации объектов перед началом решения задач. В сущности, если только вам не требуется специальная обработка, которую встроенные типы не обеспечивают, тогда почти всегда лучше применять встроенный объект, а не реализовывать собственный. Ниже перечислены причины.

- Встроенные объекты облегчают написание программ. При решении простых задач встроенные типы часто являются всем необходимым для представления структуры предметной области. Поскольку вы бесплатно получаете мощные инструменты вроде коллекций (списков) и поисковых таблиц (словарей), то можете использовать их без промедления. С помощью исключительно встроенных типов объектов Python вы можете выполнить большую работу.
- Встроенные объекты являются компонентами расширений. Для более сложных задач может понадобиться предоставить собственные объекты с применением классов Python или интерфейсов языка C. Но как вы увидите в последующих частях книги, реализуемые вручную объекты нередко строятся на основе типов, подобных спискам и словарям. Например, структура данных стека может быть реализована в виде класса, который управляет или настраивает встроенный список.

- Встроенные объекты часто эффективнее специальных структур данных. Встроенные типы Python задействуют уже оптимизированные алгоритмы для работы со структурами данных, которые реализованы на С, чтобы обеспечивать высокое быстродействие. Несмотря на возможность самостоятельного написания похожих типов объектов, вам обычно будет трудно достичь уровня производительности, предлагаемого встроенными типами объектов.
- Встроенные объекты представляют собой стандартную часть языка. В некотором смысле Python многое заимствует из языков, которые опираются на встроенные инструменты (скажем, LISP), и языков, которые полагаются на то, что реализации инструментов или фреймворков предоставит сам программист (например, C++). Хотя вы можете реализовать уникальные типы объектов Python, вам не придется делать это для того, чтобы начать работу. Более того, поскольку встроенные типы объектов Python являются стандартом, они всегда такие же; с другой стороны, патентованные фреймворки имеют тенденцию отличаться от площадки к площадке.

Другими словами, встроенные типы объектов не только облегчают программирование, но также являются более мощными и эффективными, чем большинство того, что можно создать с нуля. Независимо от того, реализуете ли вы новые типы объектов, встроенные типы объектов образуют основу каждой программы Python.

Основные типы данных Python

В табл. 4.1 приведен обзор встроенных типов объектов Python и примеров синтаксиса, используемого для кодирования их *литералов*, т.е. выражений, которые генерируют такие объекты². Если вы работали с другими языками, то некоторые типы могут показаться знакомыми; скажем, числа и строки представляют соответственно числовые и текстовые значения, а объекты файлов предлагают интерфейс для обработки реальных файлов, хранящихся на компьютере.

Таблица 4.1. Обзор встроенных типов объектов

Тип объекта	Пример литерала/создания
Числа	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Строки	'spam', "Bob's", b'a\x01c', u'sp\ xc4m'
Списки	[1, [2, 'three'], 4.5], list(range(10))
Словари	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Кортежи	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Файлы	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Множества	set('abc'), {'a', 'b', 'c'}
Прочие основные типы	Булевские значения, сами типы, None
Типы программных единиц	Функции, модули, классы (часть IV, часть V, часть VI)
Типы, связанные с реализацией	Скомпилированный код, трассировки стека (часть IV, часть VII)

² В этой книге термин *литерал* означает просто выражение, синтаксис которого генерирует объект, иногда также называемый *константой*. Обратите внимание, что термин “константа” не подразумевает, что обозначаемый им объект или переменная никогда не может изменяться (т.е. он не связан с const в C++ или понятием “неизменяемый” в Python – тема, исследуемая в разделе “Неизменяемость” далее в главе).

Однако для ряда читателей типы объектов в табл. 4.1 могут быть более универсальными и мощными, нежели то, к чему они привыкли. Например, вы обнаружите, что одни лишь списки и словари являются мощными инструментами представления данных, позволяющими избавиться от большинства работ, которые приходится выполнять для поддержки коллекций и поиска в языках более низкого уровня. Вкратце списки предоставляют упорядоченные коллекции других объектов, а словари хранят объекты по ключам; и списки, и словари могут быть вложенными, расти и уменьшаться по требованию и содержать объекты любого типа.

Присутствующие в табл. 4.1 *программные единицы* – функции, модули и классы (мы встретимся с ними в последующих частях книги) – в Python также представляют собой объекты. Они создаются посредством операторов и выражений вроде `def`, `class`, `import` и `lambda` и могут свободно передаваться между сценариями, храниться внутри других объектов и т.д. Вдобавок Python предлагает набор типов, *связанных с реализацией*, таких как объекты скомпилированного кода, которые обычно представляют больший интерес для создателей инструментов, чем для разработчиков приложений; мы исследуем типы подобного рода в последующих частях книги, хотя и менее глубоко из-за их специализированных ролей.

Несмотря на ее заголовок, табл. 4.1 в действительности нельзя считать полной, т.к. все, что мы обрабатываем в программах Python, является какой-то разновидностью объекта. Скажем, когда мы выполняем сопоставление текста с образцом в Python, то создаем объекты образцов, а когда пишем сценарий для работы с сетью, то применяем объекты сокетов. Эти другие виды объектов, как правило, создаются путем импортирования и использования функций в библиотечных модулях (`re` и `socket` для сопоставления с образцом и для сокетов) и обладают своим поведением.

Тем не менее, другие типы объектов из табл. 4.1 мы обычно называем *основными* типами данных, потому что они фактически встроены в язык Python, т.е. для генерации большинства из них имеется специфический синтаксис выражений. Например, когда вы запускаете следующий код с символами, заключенными в кавычки:

```
>>> 'spam'
```

то, говоря формально, выполняете литературное выражение, которое генерирует и возвращает новый объект *строки*. Для создания этого объекта в языке Python существует специальный синтаксис. Аналогичным образом выражение, помещенное в квадратные скобки, создает *список*, выражение в фигурных скобках – *словарь* и т.д. Хотя, как мы увидим, в Python нет объявлений типов, типы создаваемых и применяемых объектов определяет синтаксис выполняемых выражений. Фактически выражения, генерирующие объекты, такие как приведенные в табл. 4.1, в общем смысле являются источником происхождения типов в языке Python.

Не менее важно и то, что после того, как объект создан, он навсегда привязывается к своему набору операций; на строке можно выполнять только строковые операции, а на списке – только списковые операции. Формально это означает, что язык Python *динамически типизирован* – модель, которая отслеживает типы автоматически, не требуя объявления, но он также *строго типизирован* – ограничение, обусловливающее возможность выполнения на объекте только допустимых для его типа операций.

Каждый тип объекта из табл. 4.1 мы подробно исследуем в последующих главах. Однако прежде чем погружаться в детали, давайте посмотрим на основные объекты Python в действии. Далее в главе предлагается обзор операций, которые будут рассматриваться в оставшихся главах части. Не ожидайте здесь найти полную историю – цель главы в том, чтобы разжечь аппетит и представить ряд ключевых идей. Тем не менее, лучший способ начать – приступить к работе, поэтому мы сразу перейдем к реалистичному коду.

Числа

Если в прошлом вы занимались программированием или написанием сценариев, тогда некоторые типы из табл. 4.1 вероятно покажутся знакомыми. Но даже когда это не так, числа довольно-таки прямолинейны. Набор основных объектов Python включает ожидаемые типы: *целые числа*, не имеющие дробной части, числа с *плавающей точкой*, которые имеют дробную часть, и более экзотические типы – *комплексные* числа с мнимой частью, *десятичные* числа с фиксированной точностью, *рациональные* числа с числителем и знаменателем, а также полнофункциональные *множества*. Встроенных чисел вполне достаточно для представления большинства числовых величин (от вашего возраста до сальдо вашего банковского счета), но доступно еще больше типов в виде сторонних дополнений.

Несмотря на предложение ряда причудливых вариантов, основные числовые типы Python, в общем-то, являются базовыми. Числа в Python поддерживают обычные математические операции. Например, плюс (+) выполняет сложение, звездочка (*) используется для умножения, а две звездочки (**) применяются для возведения в степень:

```
>>> 123 + 222          # Целочисленное сложение
345
>>> 1.5 * 4           # Умножение с плавающей точкой
6.0
>>> 2 ** 100          # Снова 2 в степени 100
1267650600228229401496703205376
```

Обратите внимание на результат последней операции: целочисленный тип Python 3.X при необходимости автоматически обеспечивает повышенную точность для больших чисел (в Python 2.X числа, слишком большие для обычного целочисленного типа, поддерживались отдельным длинным целочисленным типом). Скажем, в Python вы можете вычислить 2 в степени 1 000 000 в виде целого числа, но вероятно не должны выводить результат – он будет содержать свыше 300 000 цифр, поэтому придется подождать:

```
>>> len(str(2 ** 1000000))    # Сколько цифр в действительно БОЛЬШОМ числе?
301030
```

Такая форма вложенных вызовов работает изнутри наружу – сначала результирующее число операции ** преобразуется в строку цифр с помощью встроенной функции str, после чего посредством len получается длина итоговой строки. Конечным результатом будет количество цифр. Функции str и len работают со многими типами объектов; в ходе чтения книги вы узнаете о них больше сведений.

В версиях, предшествующих Python 2.7 и Python 3.1, после начала экспериментирования с числами с плавающей точкой вы вполне вероятно встретите кое-что, на первый взгляд кажущееся странным:

```
>>> 3.1415 * 2          # repr: вид как в коде (Python < 2.7 и 3.1)
6.2830000000000004
>>> print(3.1415 * 2)   # str: вид, дружественный к пользователю
6.283
```

Первый результат не является ошибкой; это проблема отображения. На самом деле есть два способа вывода любого объекта в Python – с полной точностью (как в первом результате) и в форме, дружественной к пользователю (как во втором результате). Формально первая форма называется repr (вид объекта как в коде), а вторая –

`str` (вид, дружественный к пользователю). В старых версиях Python форма `repr` для чисел с плавающей точкой временами отображала с большей точностью, чем можно было ожидать. Разница также может быть значимой, когда мы подойдем к использованию классов. А пока если что-то выглядит странным, попробуйте отобразить его с помощью оператора вызова встроенной функции `print`.

Но лучше обновиться до версии Python 2.7 и последней Python 3.X, где числа с плавающей точкой отображаются более разумно, обычно с меньшим количеством паразитных цифр – поскольку книга основана на Python 2.7 и Python 3.X, такая форма отображения чисел с плавающей точкой будет встречаться повсеместно:

```
>>> 3.1415 * 2                                # repr: вид как в коде (Python >= 2.7 и 3.1)
6.283
```

Помимо выражений в состав Python входит несколько полезных числовых модулей (*модули* – это просто пакеты дополнительных инструментов, которые мы импортируем для их применения):

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

Модуль `math` содержит более сложные числовые инструменты в виде функций, а модуль `random` выполняет генерацию случайных чисел и случайный выбор (здесь из задаваемого в квадратных скобках списка Python – упорядоченной коллекции других объектов, которая будет представлена позже в главе):

```
>>> import random
>>> random.random()
0.7082048489415967
>>> random.choice([1, 2, 3, 4])
1
```

Python также включает более экзотические числовые объекты, такие как комплексные числа, числа с фиксированной точностью, рациональные числа, множества и булевские числа, а в области сторонних расширений с открытым кодом доступно еще больше числовых объектов (например, матрицы, векторы и числа с повышенной точностью). Мы обсудим их позже в главе и книге.

До сих пор мы использовали Python во многом подобно простому калькулятору; чтобы лучше оценить его встроенные типы, давайте перейдем к исследованию строк.

Строки

Строки применяются для записи текстовой информации (каждем, вашего имени) и произвольных совокупностей байтов (наподобие содержимого файла изображения). Они являются первым примером того, что в Python называется *последовательностью* – позиционно упорядоченной коллекцией других объектов. Для содержащихся элементов последовательности поддерживают порядок слева направо: элементы сохраняются и извлекаются по своим относительным позициям. Строго говоря, строки представляют собой последовательности односимвольных строк; другие более универсальные типы последовательностей включают списки и кортежи, рассматриваемые далее.

Операции над последовательностями

Как последовательности, строки поддерживают операции, которые предполагают наличие позиционного порядка среди элементов. Например, если мы имеем четырехсимвольную строку, записанную в кавычках (обычно одинарных), то можем проверить ее длину с помощью встроенной функции `len` и извлечь ее компоненты посредством выражений *индексации*:

```
>>> S = 'Spam'      # Создать 4-символьную строку и присвоить ее некоторому имени
>>> len(S)          # Длина
4
>>> S[0]            # Первый элемент в S, который индексируется по позиции,
                   # начиная с нуля
'S'
>>> S[1]            # Второй элемент слева
'p'
```

Индексы в Python представляют собой смещения спереди и потому начинаются с 0: первый элемент находится по индексу 0, второй — по индексу 1 и т.д.

Обратите внимание, как здесь строка присваивается *переменной* по имени `S`. Деталями работы присваивания мы займемся позже (главным образом в главе 6), но переменные Python никогда не нужно объявлять заранее. Переменная создается, когда вы присваиваете ей значение, которое может быть объектом любого типа, и она заменяется присвоенным значением, когда появляется в выражении. Переменной должно быть что-то присвоено ко времени использования ее значения. В настоящей главе достаточно знать, что для сохранения объекта с целью дальнейшей работы с ним необходимо присвоить его *переменной*.

В Python мы можем также индексировать в обратном направлении, с конца, т.е. положительные индексы считаются слева, а отрицательные — справа:

```
>>> S[-1]           # Последний элемент с конца в S
'm'
>>> S[-2]           # Второй элемент с конца в S
'a'
```

Формально отрицательный индекс добавляется к длине строки, так что следующие две операции эквивалентны (хотя первую проще записывать и менее легко допустить ошибку):

```
>>> S[-1]           # Последний элемент в S
'm'
>>> S[len(S)-1]    # Отрицательная индексация, сложный путь
'm'
```

Обратите внимание, что в квадратных скобках мы можем применять *произвольное выражение*, а не только жестко закодированный числовой литерал — везде, где Python ожидает значение, допускается использовать литерал, переменную или любое желаемое выражение. В этом отношении синтаксис Python полностью универсален.

В дополнение к простой позиционной индексации последовательности также поддерживают более общую форму индексации, известную как *нарезание*, которое представляет собой способ извлечения целой части (реза) за один шаг. Вот пример:

```
>>> S               # 4-символьная строка
'Spam'
>>> S[1:3]          # Срез S со смещения 1 до 2 (не 3)
'pa'
```

Вероятно, проще всего думать о срезах как о способе извлечения из строки целого *раздела* за один шаг. Универсальная форма среза, $X[I:J]$, означает “предоставить из X все содержимое, начиная со смещения I и заканчивая смещением J, не включая его”. Результат возвращается в новом объекте. Вторая из приведенных выше операций выдает все символы из строки S со смещения 1 до 2 (т.е. со смещения 1 до 3-1) в виде новой строки. Следствием оказывается нарезание или “выборка” двух символов из середины.

По умолчанию левая граница среза принимается равной нулю, а правая — длине нарезаемой последовательности. Это приводит к нескольким распространенным вариантам применения:

```
>>> S[1:]      # Все после первого элемента (1:len(S))
'Spam'
>>> S          # Сама строка S не изменилась
'Spam'
>>> S[0:3]     # Все кроме последнего элемента
'Spa'
>>> S[:3]      # То же, что и S[0:3]
'Spa'
>>> S[:-1]     # Снова все кроме последнего элемента, но проще (0:-1)
'Spa'
>>> S[:]       # Вся строка S как копия верхнего уровня (0:len(S))
'Spam'
```

Во второй с конца операции демонстрируется возможность использования отрицательных смещений в границах срезов, а последняя операция фактически копирует целую строку. Как вы узнаете позже, копировать строку не имеет смысла, но такая форма может оказаться полезной в последовательностях, подобных спискам.

Наконец, будучи последовательностями, строки поддерживают также *конкатенацию* (объединение двух строк в новую строку), обозначаемую знаком “плюс”, и *повторение* (создание новой строки путем повторения другой):

```
>>> S
'Spam'
>>> S + 'xyz'    # Конкатенация
'Spamxyz'
>>> S           # S не изменяется
'Spam'
>>> S * 8        # Повторение
'SpamSpamSpamSpamSpamSpamSpam'
```

Важно отметить, что знак “плюс” (+) для разных объектов означает отличающиеся действия: сложение для чисел и конкатенацию для строк. Это общее свойство Python позже в книге мы назовем *полиморфизмом* — коротко говоря, смысл операции зависит от объектов, к которым она применяется. Во время изучения динамической типизации вы увидите, что свойство полиморфизма объясняет большую часть причин лаконичности и гибкости кода Python. Поскольку типы не ограничены, операция Python способна нормально работать со многими разными типами объектов автоматически при условии, что они поддерживают совместимый интерфейс (подобно операции + здесь). Полиморфизм — крупное понятие в Python; вы узнаете о нем далее в книге.

Неизменяемость

В предшествующих примерах также обратите внимание на то, что выполнение всех операций не вызывало изменения исходной строки. Каждая строковая операция определена так, чтобы производить в качестве результата новую строку, потому что строки в Python являются *неизменяемыми* – после создания их нельзя модифицировать на месте. Другими словами, вы никогда неerezапишете значения неизменяемых объектов. Например, вы не сумеете изменить строку, присваивая значение символу в одной из ее позиций, но всегда сможете построить новую строку и назначить ей то же самое имя. С учетом того, что в ходе работы Python очищает старые объекты (как вы увидите позже), это не настолько неэффективно, как может показаться:

```
>>> S
'Spam'
>>> S[0] = 'z'          # Неизменяемые объекты модифицировать нельзя
...текст сообщения об ошибке не показан...
TypeError: 'str' object does not support item assignment
Ошибка типа: объект str не поддерживает присваивание в отношении элементов
>>> S = 'z' + S[1:]  # Но мы можем выполнять выражения для создания новых объектов
>>> S
'zspam'
```

Каждый объект в Python классифицируется как *неизменяемый* (немодифицируемый) или нет. Что касается основных типов, то *числа, строки и кортежи* неизменяемы, а *списки, словари и множества* – нет; они могут свободно модифицироваться на месте, как большинство новых объектов, которые вы будете создавать с помощью классов. Такое отличие оказывается критически важным в работе Python, но мы пока не в состоянии полностью исследовать его влияние. Помимо прочего неизменяемость можно использовать для гарантирования того, что объект остается постоянным на протяжении всей программы; значения изменяемых объектов способны меняться в любой момент и в любом месте (ожидаете вы этого или нет).

Строго говоря, текстовые данные можно изменять *на месте*, если развернуть их в *список* индивидуальных символов и объединить вместе с пустым разделителем или применить более новый тип `bytearray`, доступный в Python 2.6, 3.0 и последующих версиях:

```
>>> S = 'shrubbery'
>>> L = list(S)                      # Развернуть в список: [...]
>>> L
['s', 'h', 'r', 'u', 'b', 'b', 'e', 'r', 'y']
>>> L[1] = 'c'                        # Изменить на месте
>>> ''.join(L)                      # Объединить с пустым разделителем
'scrubbery'

>>> B = bytearray(b'spam')
      # Гибрид байтов/списка
>>> B.extend(b'eggs')                # b необходимо в Python 3.X, но не в Python 2.X
>>> B
      # B[i] = ord(x) тоже здесь работает
bytearray(b'spameggs')
>>> B.decode()                      # Преобразовать в обычную строку
'spameggs'
```

Тип `bytearray` поддерживает изменения на месте для текста, но только в отношении символов, которые имеют ширину не более 8 битов (скажем, ASCII). Все остальные строки по-прежнему неизменяемы – `bytearray` представляет собой особый гибрид неизменяемых *байтовых* строк (с синтаксисом `b'...'`, обязательным в Python 3.X и необязательным в Python 2.X) и изменяемых *списков* (обозначаемых и отображаемых

в `[]`). Чтобы полностью понять приведенный выше код, мы должны узнать больше о них и тексте Unicode.

Методы, специфичные для типа

Каждая строковая операция, изученная до сих пор, на самом деле является операцией над последовательностью, т.е. такие операции будут работать также с другими последовательностями в Python, включая списки и кортежи. Однако в дополнение к универсальным операциям над последовательностями строки также имеют собственные операции, доступные в виде *методов* – функций, которые присоединены и действуют на специфическом объекте и инициируются посредством выражений вызовов.

Например, строковый метод `find` представляет собой операцию поиска подстроки (она возвращает смещение указанной подстроки или `-1`, если подстрока не найдена), а строковый метод `replace` выполняет глобальный поиск и замену; оба метода действуют на объекте, к которому они присоединены и вызваны:

```
>>> S = 'Spam'  
>>> S.find('pa')                      # Найти смещение подстроки в S  
1  
>>> S  
'Spam'  
>>> S.replace('pa', 'XYZ')           # Заменить вхождения подстроки в S другой подстрокой  
'SXYZm'  
>>> S  
'Spam'
```

Невзирая на имена этих строковых методов, они не изменяют исходные строки, а создают в качестве результатов новые строки – из-за того, что строки неизменяемы, методы могут работать только таким способом. Строковые методы являются первой линейкой инструментов обработки текста в Python. Другие методы разбивают строку на подстроки по разделителю (удобны как простая форма разбора), выполняют преобразования регистра символов, проверяют содержимое строки (цифры, буквы и т.д.) и удаляют пробельные символы из концов строки:

```
>>> line = 'aaa,bbb,ccccc,dd'  
>>> line.split(',')                 # Разбить по разделителю в список подстрок  
['aaa', 'bbb', 'ccccc', 'dd']  
  
>>> S = 'spam'  
>>> S.upper()                      # Преобразовать в верхний и нижний регистры  
'SPAM'  
>>> S.isalpha()                    # Проверить содержимое: isalpha, isdigit и т.д.  
True  
  
>>> line = 'aaa,bbb,ccccc,dd\n'  
>>> line.rstrip()                  # Удалить пробельные символы с правой стороны  
'aaa,bbb,ccccc,dd'  
>>> line.rstrip().split(',')      # Скомбинировать две операции  
['aaa', 'bbb', 'ccccc', 'dd']
```

Взгляните на последнюю команду – она удаляет пробельные символы до того, как разбивает строку, потому что Python выполняет операции слева направо, попутно создавая временный результат. Строки также поддерживают более сложную операцию подстановки, известную как *форматирование*, которая доступна в форме выражения (изначально) и вызова строкового метода (начиная с Python 2.6 и 3.0); в Python 2.7/3.1 и последующих версиях второй вариант позволяет не указывать относительные номера аргументов:

```

>>> '%s, eggs, and %s' % ('spam', 'SPAM!')           # Выражение форматирования
# (все версии)
'spam, eggs, and SPAM!'
>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!')    # Метод форматирования
# (2.6+, 3.0+)
'spam, eggs, and SPAM!'
>>> '{}, eggs, and {}'.format('spam', 'SPAM!')      # Номера необязательны
# (2.7+, 3.1+)
'spam, eggs, and SPAM!'

```

Форматирование богато возможностями, обсуждение которых мы отложим до будущих глав книги и которые особенно важны при генерации числовых отчетов:

```

>>> '{:.2f}'.format(296999.2567)                  # Разделители, десятичные цифры
'296,999.26'
>>> '%.2f | %+05d' % (3.14159, -42)            # Цифры, дополнение, знаки
'3.14 | -0042'

```

Здесь следует сделать одно замечание: хотя операции над последовательностями являются универсальными, методы – нет; несмотря на то, что некоторые типы совместно используют ряд имен методов, строковые методы обычно работают только на строках и ни на чем другом. В качестве эмпирического правила запомните, что инструментальный набор Python делится на уровни: универсальные операции, охватывающие множество типов, выглядят как встроенные функции или выражения (например, `len(X)`, `X[0]`), но операции, специфичные для типа, представляют собой вызовы методов (скажем, `aString.upper()`). По мере все большего освоения Python поиск необходимых инструментов в этих категориях станет более естественным, а в следующем разделе даются советы, которыми вы можете воспользоваться прямо сейчас.

Получение справки

Представленные в предыдущем разделе методы – репрезентативная, но небольшая выборка того, что доступно для объектов строк. В целом эта книга не является исчерпывающей в плане рассмотрения методов объектов. Для получения дополнительных деталей вы всегда можете вызвать встроенную функцию `dir`. Когда функция `dir` вызывается без аргументов, она выводит список переменных, присвоенных в области видимости вызывающего объекта. Более полезно то, что функция `dir` возвращает список всех атрибутов, доступных для любого переданного ей объекта. Поскольку методы являются функциональными атрибутами, они будут присутствовать в таком списке. Предполагая, что `S` – все еще строка, вот ее атрибуты в Python 3.3 (в Python 2.X они немного отличаются):

```

>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', 'capitalize', 'casfold', 'center', 'count',
 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

```

Скорее всего, имена с *парами символов подчеркивания* вас не будут заботить вплоть до момента, когда начнется обсуждение перегрузки операций в классах – они представляют реализацию объекта строки и предназначены для поддержки настройки. Например, метод `__add__` объектов строк является тем, который в действительности выполняет конкатенацию; внутренне Python отображает первый вариант из показанных ниже на второй, хотя обычно сами вы не должны применять второй вариант (он менее понятен и может даже выполнятся медленнее):

```
>>> S + 'NI!'
'spamNI!'
>>> S.__add__('NI!')
'spamNI!'
```

Вообще говоря, ведущие и завершающие пары символов подчеркивания – это шаблон именования, используемый Python для деталей реализации. Имена без символов подчеркивания в списке, выведенном функцией `dir`, являются вызываемыми методами на объектах строк.

Функция `dir` просто выдает имена методов. Чтобы выяснить, что делает тот или иной метод, его имя можно передать функции `help`:

```
>>> help(S.replace)
Help on built-in function replace:
Справка по встроенной функции replace:

replace(...)
    S.replace(old, new[, count]) -> str
    Return a copy of S with all occurrences of substring
    old replaced by new. If the optional argument count is
    given, only the first count occurrences are replaced.
    Возвращает копию S, в которой все вхождения подстроки
    old заменены подстрокой new. Если необязательный аргумент
    задан, тогда заменяются только первые count вхождений.
```

Функция `help` – один из удобных интерфейсов к поставляемой вместе с Python системе кода, которая называется *PyDoc* и представляет собой инструмент для извлечения документации из объектов. Позже в книге вы увидите, что PyDoc способна также визуализировать свои отчеты в формате HTML с целью отображения в веб-браузере.

Вы можете также запросить справку по *целой строке* (например, `help(S)`), но спрашивающих сведений может быть больше или меньше, чем вы хотели видеть. В более старых версиях Python выводится информация о каждом строковом методе, а в новых версиях Python вообще ничего не выводится, т.к. строки трактуются специальным образом. Обычно лучше запрашивать справку по специальному *методу*.

Функции `dir` и `help` также принимают в качестве аргументов либо реальный *объект* (подобный строке `S`), либо имя *типа данных* (вроде `str`, `list` и `dict`). Последняя форма возвращает тот же самый список для `dir`, но отображает полные сведения о типе для `help`, и позволяет запрашивать справку по специальному методу через имя типа (например, справку по `str.replace`).

За дополнительными деталями обращайтесь в справочное руководство по стандартной библиотеке Python или в другие справочники, но `dir` и `help` представляют собой первый уровень документации в Python.

Другие способы написания строк

До сих пор мы рассматривали операции над последовательностями и методы, специфичные для типа, объекта строки. В Python также предлагаются разнообраз-

ные способы написания строк, которые мы более глубоко исследуем далее в книге. Скажем, специальные символы могут быть представлены как управляющие последовательности, начинающиеся с обратной косой черты, которые Python отображает с помощью шестнадцатеричной системы обозначений `\xNN`, если они не представляют печатаемые символы:

```
>>> s = 'A\nB\tC'      # \n - конец строки, \t - табуляция
>>> len(s)            # Как \n, так и \t является только одним символом
5
>>> ord('\n')          # \n - один символ, кодируемый как десятичное значение 10
10
>>> s = 'A\0B\0C'      # \0, байт с двоичными нулями, не завершает строку
>>> len(s)
5
>>> s                  # Непечатаемые символы отображаются как шестнадцатеричные
# управляющие последовательности \xNN
'A\x00B\x00C'
```

Строки в Python разрешено заключать в *одинарные* или *двойные* кавычки – они обозначают то же самое, но позволяют встраивать кавычки другого вида без отмены (большинство программистов отдают предпочтение одинарным кавычкам). Кроме того, Python дает возможность заключать многострочные строковые литералы в *утроенные* кавычки (одинарные или двойные). Когда применяется такая форма, все строки объединяются вместе с добавлением символов конца строки там, где встречаются разрывы строки. Хотя это мелкое синтаксическое удобство, оно полезно при встраивании в сценарий Python таких вещей, как многострочный код HTML, XML или JSON, и временной заглушки кода – нужно просто поместить три кавычки до и после фрагмента кода:

```
>>> msg = """
aaaaaaaaaaaaaa
bbb'''bbbbbbbbb'''bbbbbbbbb'bbbb
cccccccccccccc
"""

>>> msg
'\naaaaaaaaaaaa\nbbb\''\''\''bbbbbbbbb'''bbbbbbbbb\''bbb\nccccccccccccccc\''\n'
```

Вдобавок Python поддерживает *неформатированные* строковые литералы, которые отключают механизм отмены посредством обратной косой черты. Такие литералы начинаются с буквы `r` и полезны для представления строк, подобных путям к каталогам в Windows (например, `r'C:\text\new'`).

Строки Unicode

Строки Python также сопровождаются полной поддержкой *Unicode*, требующейся для обработки текста в интернационализированных наборах символов. Скажем, символы в японском и русском алфавитах находятся за пределами набора ASCII. Такой текст, отличающийся от ASCII, может появляться на веб-страницах, в сообщениях электронной почты, внутри графических пользовательских интерфейсов, JSON, XML или где-то еще. Правильная обработка подобного текста требует поддержки Unicode, которая встроена в Python, но ее форма варьируется от версии к версии и относится к самым заметным различиям.

В Python 3.X обычная строка `str` поддерживает текст Unicode (в том числе набор ASCII, который является подмножеством Unicode); отдельный строковый тип `bytes`

представляет неформатированные байтовые значения (включая закодированный текст); и в целях совместимости литералы Unicode из Python 2.X поддерживаются в Python 3.3 и последующих версиях (они трактуются так же, как обычные строки str из Python 3.X):

```
>>> 'sp\xc4m'           # Python 3.X: обычные строки str являются текстом Unicode
'spAm'
>>> b'a\x01c'          # Строки bytes - это данные, основанные на байтах
b'a\x01c'
>>> u'sp\u00c4m'        # Литералы Unicode из Python 2.X работают в Python 3.3+:
# просто строка str
'spAm'
```

В Python 2.X обычная строка str поддерживает строки 8-битных символов (в том числе текст ASCII) и неформатированные байтовые значения; отдельный строковый тип unicode представляет текст; в целях совместимости байтовые литералы Python 3.X поддерживаются в Python 2.6 и последующих версиях (они трактуются так же, как обычные строки str из Python 2.X):

```
>>> print u'sp\xc4m' # Python 2.X: строки Unicode являются отдельным типом
spam
>>> 'a\x01c'           # Обычные строки str содержат текст/данные,
# основанные на байтах
'a\x01c'
>>> b'a\x01c'          # Байтовые литералы из Python 3.X работают в Python 2.6+:
# просто строка str
'a\x01c'
```

Формально в Python 2.X и 3.X строки, отличающиеся от Unicode, представляют собой последовательности *8-битных байтов*, которые выводятся в виде символов ASCII, когда это возможно, а строки Unicode – последовательности *кодовых точек Unicode* (идентифицирующих чисел для символов, которые вовсе не обязательно отображаются на одиночные байты при кодировании в файлах или хранении в памяти). На самом деле понятие байтов неприменимо к Unicode: определенные кодировки содержат кодовые точки, слишком большие, чтобы уместиться в байт, и даже простой текст в 7-битном ASCII не выделяет один байт на символ для ряда кодировок и схем хранения в памяти:

```
>>> 'spam'              # Символы могут занимать 1, 2 или 4 байта в памяти
'spam'
>>> 'spam'.encode('utf8') # В UTF-8 кодируется как 4 байта в файлах
b'spam'
>>> 'spam'.encode('utf16') # Но в UTF-16 кодируется как 10 байтов
b'\xff\xfe\x00p\x00a\x00m\x00'
```

Версии Python 3.X и 2.X поддерживают и встречавшийся ранее строковый тип bytearray, который по существу является строкой bytes (строкой str в Python 2.X), поддерживающей большинство операций изменения на месте объекта списка.

В версиях Python 3.X и 2.X поддерживается кодирование символов, *отличающихся от ASCII*, с помощью шестнадцатеричной \x, короткой \u и длинной \U управляющих последовательностей Unicode, а также файловых кодировок, которые объявлены в файлах исходного кода программы. Вот три способа кодирования символа, отличающегося от ASCII, в Python 3.X (чтобы увидеть то же самое в Python 2.X, добавьте ведущий символ u и print):

```
>>> 'sp\xc4\u00c4\U0000000c4m'
'spAAAm'
```

Смысл этих значений и способы их использования отличаются между *текстовыми строками*, которые являются обычными строками в Python 3.X и Unicode в Python 2.X, и *байтовыми строками*, которые представляют собой байты в Python 3.X и обычные строки в Python 2.X. Для внедрения действительных порядковых целочисленных значений кодовых точек Unicode в текстовые строки могут применяться все управляющие последовательности. Напротив, для внедрения закодированной формы текста в байтовые строки используются только шестнадцатеричные управляющие последовательности \x, но не декодированные значения кодовых точек – декодированные байты совпадают с кодовыми точками лишь для некоторых кодировок и символов:

```
>>> '\u00A3', '\u00A3'.encode('latin1'), b'\xa3'.decode('latin1')
('£', b'\xa3', '£')
```

Заметная разница заключается в том, что Python 2.X позволяет смешивать в выражениях обычные строки и строки Unicode при условии, что обычные строки представлены в кодировке ASCII; в противоположность этому Python 3.X имеет более строковую модель, которая *никогда* не разрешает смешивать обычные и байтовые строки без явного преобразования:

u'x' + b'y'	# Работает в Python 2.X (где символ b необязателен # и игнорируется)
u'x' + 'y'	# Работает в Python 2.X: u'xy'
u'x' + b'y'	# Не работает в Python 3.3 # (где символ и необязателен и игнорируется)
u'x' + 'y'	# Работает в Python 3.3: 'xy'
'x' + b'y'.decode()	# Работает в Python 3.X, если декодировать # байты в строку: 'xy'
'x'.encode() + b'y'	# Работает в Python 3.X, если закодировать # строку в байты: b'xy'

Помимо рассмотренных строковых типов обработка Unicode сводится главным образом к передаче текстовых данных в и из *файлов* – текст *кодируется* в байты, когда сохраняется в файле, и декодируется в символы (кодовые точки), когда читается обратно в память. После загрузки текст обычно обрабатывается как строки только в декодированной форме.

Тем не менее, из-за такой модели в Python 3.X файлы также зависят от содержимого: *текстовые файлы* реализуют именованные кодировки, принимают и возвращают строки *str*, но *двоичные файлы* взамен имеют дело со строками *bytes* для неформатированных двоичных данных. В Python 2.X обычным содержимым файлов являются байты *str*, а специальный модуль *codecs* обрабатывает Unicode и представляет содержимое с помощью типа *unicode*.

Далее в главе мы еще вернемся к применению кодировки Unicode в файлах и обсудим ее позже в книге. Она будет кратко затрагиваться в примере с символами валют, рассматриваемом в главе 25, но более подробно исследуется в части книги, посвященной более сложным темам. Кодировка Unicode важна в ряде областей, но многим программистам достаточно лишь беглого знакомства с ней. Если вы работаете только с текстом ASCII, тогда положение дел со строками и файлами по большому счету одинаково в Python 2.X и 3.X. Если же вы – новичок в программировании, то можете спокойно отложить изучение большинства деталей Unicode до тех пор, пока не овладеете основами использования строк.

Сопоставление с образцом

Прежде чем продолжить, полезно отметить, что ни один из собственных методов объекта строки не поддерживает обработку текста, основанную на образцах. Сопоставление текста с образцом представляет собой расширенный инструмент, который выходит за рамки тематики настоящей книги, но читателям, имеющим опыт работы с другими языками написания сценариев, возможно, небезынтересно будет узнать, что для выполнения сопоставления с образцом в Python мы импортируем модуль по имени `re`. Модуль `re` имеет похожие вызовы для поиска, разбиения и замены, но поскольку при указании подстрок допускается применять образцы, мы можем расчитывать на большую универсальность:

```
>>> import re
>>> match = re.match('Hello[ \t]*(.*)world', 'Hello      Python world')
>>> match.group(1)
'Python '
```

В примере производится поиск подстроки, начинающейся со слова `Hello`, за которым следует ноль или более табуляций либо пробелов, затем произвольные символы, подлежащие сохранению в качестве группы совпадения, и завершаются они словом `world`. Если такая подстрока будет найдена, тогда ее части, которые соответствуют частям образца, заключенным в круглые скобки, окажутся доступными в виде групп. Например, следующий образец выбирает три группы, разделенные символами косой черты или двоеточия:

```
>>> match = re.match('[/:](.*)[/:](.*)[/:](.*), '/usr/home:lumberjack')
>>> match.groups()
('usr', 'home', 'lumberjack')
>>> re.split('[/:]', '/usr/home/lumberjack')
 ['', 'usr', 'home', 'lumberjack']
```

Сопоставление с образцом само по себе является расширенным инструментом обработки текста, но в Python имеется поддержка даже еще более сложной обработки текста и языков, включая разбор XML, HTML и анализ естественного языка. Краткие примеры образцов и разбора XML будут приведены в конце главы 37, а мы заканчиваем со строками и переходим к следующему типу.

Списки

Объект списка Python является наиболее общей *последовательностью*, предлагаемой языком. Списки представляют собой позиционно упорядоченные коллекции объектов произвольных типов и не имеют фиксированных размеров. Кроме того, они *изменяются* — в отличие от строк списки можно модифицировать на месте путем присваивания по смещениям и вызова разнообразных списковых методов. Соответственно они предоставляют очень гибкий инструмент для представления произвольных коллекций — перечня файлов в каталоге, сотрудников в компании, сообщений в ящике входящей почты и т.д.

Операции над последовательностями

Будучи последовательностями, списки поддерживают все операции над последовательностями, которые мы обсуждали для строк; единственное отличие в том, что результатами обычно будут не строки, а списки. Например, имея список из трех элементов:

```
>>> L = [123, 'spam', 1.23]      # Список из трех объектов разных типов
>>> len(L)                      # Количество элементов в списке
3
```

мы можем его индексировать, нарезать и выполнять другие действия в точности, как поступали со строками:

```
>>> L[0]                         # Индексация по позиции
123
>>> L[:-1]                        # Нарезание списка возвращает новый список
[123, 'spam']
>>> L + [4, 5, 6]                 # Конкатенация и повторение также создают новые списки
[123, 'spam', 1.23, 4, 5, 6]
>>> L * 2
[123, 'spam', 1.23, 123, 'spam', 1.23]
>>> L                             # Исходный список не изменился
[123, 'spam', 1.23]
```

Операции, специфичные для типа

Списки Python могут напоминать *массивы* в других языках, но они, как правило, более мощные. Прежде всего, отсутствует ограничение, что элементы должны принадлежать какому-то фиксированному *типу* — скажем, показанный выше список содержал три объекта совершенно разных типов (целое число, строку и число с плавающей точкой). Кроме того, списки не имеют фиксированных *размеров*. То есть они могут увеличиваться и уменьшаться в ответ на операции, специфичные для списков:

```
>>> L.append('NI')    # Увеличение: добавление объекта в конец списка
>>> L
[123, 'spam', 1.23, 'NI']
>>> L.pop(2)          # Уменьшение: удаление элемента из середины
1.23
>>> L                # del L[2] также выполняет удаление из списка
[123, 'spam', 'NI']
```

Списковый метод `append` увеличивает размер списка и помещает объект в конец; метод `pop` (или эквивалентный оператор `del`) удаляет элемент по заданному смещению, приводя к уменьшению списка. Другие списковые методы вставляют объект в произвольную позицию (`insert`), удаляют указанный элемент по значению (`remove`), добавляют множество элементов в конец (`extend`) и т.д. Поскольку списки являются изменяемыми, большинство списковых методов также модифицируют объект списка на месте, а не создают новый такой объект:

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

Например, метод `sort` по умолчанию упорядочивает список по возрастанию, а метод `reverse` обращает его — в обоих случаях методы модифицируют список напрямую.

Контроль границ

Хотя списки не имеют фиксированных размеров, Python все же не разрешает ссылаться на несуществующие элементы.

Индексация и присваивание за концом списка всегда считается ошибкой:

```
>>> L
[123, 'spam', 'NI']
>>> L[99]
...текст сообщения об ошибке не показан...
IndexError: list index out of range
Ошибка индекса: индекс в списке выходит за допустимые пределы
>>> L[99] = 1
...текст сообщения об ошибке не показан...
IndexError: list assignment index out of range
IndexError: индекс присваивания в списке выходит за допустимые пределы
```

Это сделано намеренно, т.к. попытка присваивания за концом списка обычно является ошибкой (особенно неприятной в языке C, где не выполняется столько проверок на предмет ошибок, как в Python). Вместо того чтобы в ответ молча увеличивать список, Python сообщает об ошибке. Для увеличения списка необходимо вызывать списковые методы наподобие `append`.

Вложение

Одна приятная особенность основных типов данных Python заключается в том, что они поддерживают произвольное *вложение* – мы можем вкладывать их в любой комбинации и на любую желаемую глубину. Скажем, у нас может быть список, который содержит словарь, содержащий еще один список, и т.д. Одно из прямых применений такой возможности связано с представлением в Python матриц, или “многомерных массивов”. Список с вложенными списками пригоден для базовых приложений (в строках 2 и 3 вы получите приглашение ... при работе в некоторых интерфейсах, но не в IDLE):

```
>>> M = [[1, 2, 3],    # Матрица 3 x 3 в виде вложенных списков
           [4, 5, 6],    # При использовании квадратных скобок код
           [7, 8, 9]]    # может занимать несколько строк
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Здесь мы создаем список, который содержит три других списка. Результатом будет представление матрицы 3×3 чисел. Доступ к такой структуре может осуществляться разнообразными способами:

```
>>> M[1]                # Получить строку 2
[4, 5, 6]
>>> M[1][2]              # Получить строку 2 и затем элемент 3 внутри этой строки
6
```

Первая операция извлекает вторую строку целиком, а вторая – третий элемент данной строки (она выполняется слева направо, как ранее применявшаяся операции `strip` и `split` для строк). Увязывание вместе операций индексации позволяет нам перемещаться все глубже и глубже в структуру вложенных объектов³.

³ Такая матричная структура подходит для решения небольших задач, но для более серьезных математических расчетов вероятно лучше использовать одно из численных расширений Python вроде систем с открытым кодом *NumPy* и *SciPy*. Инструменты подобного рода способны хранить и обрабатывать крупные матрицы гораздо эффективнее, чем структура с вложенными списками. Говорят, что *NumPy* превращает Python в эквивалент бесплатной и более мощной версии системы *Matlab*, и такие организации, как НАСА, Лос-Аламос, JPL и многие другие, применяют этот инструмент для решения научных и финансовых задач. Ищите подробности в веб-сети.

Списковые включения

В дополнение к операциям над последовательностями и списковым методам в Python имеется более сложная операция, известная как *выражение списка включения* (*list comprehension*), которая оказывается мощным способом обработки структур, подобных нашей матрице. Предположим, например, что необходимо извлечь второй столбец из матрицы. Получать строки легко посредством простой индексации, потому что матрица хранится по строкам, но получить столбец почти так же легко с помощью спискового включения:

```
>>> col2 = [row[1] for row in M]      # Собрать элементы в столбце 2
>>> col2
[2, 5, 8]
>>> M                                # Матрица не изменилась
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Списковые включения происходят от системы обозначения множеств; они являются способом построения нового списка за счет выполнения выражения на каждом элементе в последовательности, по одному за раз, слева направо. Списковые включения записываются в квадратных скобках (намекая на тот факт, что они создают список) и состоят из выражения и циклической конструкции, которые разделяют имя переменной (здесь `row`). Предыдущее списковое включение означает “предоставить `row[1]` из каждой строки матрицы `M` в новом списке”. Результатом будет новый список, содержащий столбец 2 матрицы.

На практике списковые включения могут быть более сложными:

```
>>> [row[1] + 1 for row in M]      # Добавить 1 к каждому элементу в столбце 2
[3, 6, 9]
>>> [row[1] for row in M if row[1] % 2 == 0]  # Отфильтровать нечетные элементы
[2, 8]
```

Первая операция добавляет 1 к каждому элементу по мере их накопления, а вторая использует конструкцию `if` для фильтрации нечетных чисел из результата с применением выражения целочисленного деления (%). Списковые включения создают новые списки результатов, но они могут использоваться для прохода по любому *итерируемому* объекту — термин, который мы конкретизируем позже. Например, ниже списковые включения применяются для прохода по жестко закодированному списку координат и по строке:

```
>>> diag = [M[i][i] for i in [0, 1, 2]]    # Собрать диагональ из матрицы
>>> diag
[1, 5, 9]
>>> doubles = [c * 2 for c in 'spam']        # Повторить символы в строке
>>> doubles
['ss', 'pp', 'aa', 'mm']
```

Такие выражения также могут использоваться для сбора множества значений при условии их помещения во вложенную коллекцию. Ниже демонстрируется применение `range` — встроенной функции, которая генерирует последовательные целые числа и в Python 3.X для отображения всех своих значений требует наличия окружающего вызова `list` (в Python 2.X одновременно создается физический список):

```
>>> list(range(4))                  # 0..3 (в Python 3.X требуется вызов list())
[0, 1, 2, 3]
>>> list(range(-6, 7, 2))          # от -6 до +6 с шагом 2
# (в Python 3.X нужен вызов list())
[-6, -4, -2, 0, 2, 4, 6]
```

```
>>> [[x ** 2, x ** 3] for x in range(4)]      # Множество значений, фильтры if
[[0, 0], [1, 1], [4, 8], [9, 27]]
>>> [[x, x / 2, x * 2] for x in range(-6, 7, 2) if x > 0]
[[2, 1, 4], [4, 2, 8], [6, 3, 12]]
```

Как вы, по всей видимости, можете сказать, списковые включения и родственные им встроенные функции вроде `map` и `filter` слишком запутаны, чтобы их удалось полностью раскрыть в обзорной главе. Главная цель этого краткого введения – показать, что в арсенале Python имеются и простые, и сложные инструменты. Списковые включения относятся к необязательным средствам, но оказываются очень удобными на практике и часто предлагают преимущество высокой скорости обработки. Они работают с любым типом, являющимся последовательностью в Python, а также с рядом типов, которые последовательностями не являются. В ходе чтения книги вы еще неоднократно столкнетесь с ними.

Однако, забегая несколько вперед, следует отметить, что в последних версиях Python синтаксис списковых включений обобщен для исполнения других ролей: в наши дни они предназначены не только для построения списков. Например, поместив списоковую включение в *круглые скобки*, можно создавать *генераторы*, которые производят результаты по требованию. В целях иллюстрации встроенная функция `sum` суммирует элементы в последовательности; вот пример суммирования всех элементов в строках нашей матрицы по требованию:

```
>>> G = (sum(row) for row in M) # Создать генератор сумм элементов в строках
>>> next(G)                  # iter(G) здесь не требуется
6
>>> next(G)                  # Запустить протокол итерации next()
15
>>> next(G)
24
```

Встроенная функция `map` способна выполнять похожую работу, генерируя результаты прогона элементов через функцию, по одному за раз и по запросу. Подобно `range` ее помещение внутрь вызова `list` приводит к возвращению всех значений в Python 3.X. В версии Python 2.X в этом нет необходимости, т.к. там функция `map` одновременно создает список результатов. Нет нужды вызывать `list` и в других контекстах, которые выполняют итерацию автоматически, если только не требуется множество проходов или поведение как у списков:

```
>>> list(map(sum, M))        # Отобразить sum на элементы в M
[6, 15, 24]
```

В версиях Python 2.7 и 3.X синтаксис списковых включений можно также использовать для создания *множеств и словарей*:

```
>>> {sum(row) for row in M}    # Создать множество сумм элементов в строках
{24, 6, 15}
>>> {i : sum(M[i]) for i in range(3)}   # Создать таблицу ключей/
                                                # значений сумм элементов в строках
{0: 6, 1: 15, 2: 24}
```

На самом деле в Python 3.X и 2.7 списки, множества, словари и генераторы могут быть простроены с помощью списковых включений:

```
>>> [ord(x) for x in 'spaaam'] # Список порядковых чисел для символов
[115, 112, 97, 97, 109]
>>> {ord(x) for x in 'spaaam'} # Множество с удаленными дубликатами
{112, 97, 115, 109}
```

```
>>> {x: ord(x) for x in 'spaam'}      # Словарь с уникальными ключами
{'p': 112, 'a': 97, 's': 115, 'm': 109}
>>> (ord(x) for x in 'spaam')        # Генератор значений
<generator object <genexpr> at 0x0000000000254DAB0>
```

Тем не менее, чтобы понять объекты, подобные генераторам, множествам и словарям, мы должны двигаться вперед.

Словари

Словари Python – нечто совершенно иное; они вообще не являются последовательностями и взамен известны как *отображения*. Отображения также представляют собой коллекции других объектов, но они хранят объекты по ключам, а не по относительным позициям. В действительности отображения не поддерживают какой-либо надежный порядок слева направо; они просто отображают ключи на связанные значения. Словари – единственный тип отображения в наборе основных объектов Python – являются *изменяемыми*: как и списки, их можно модифицировать на месте и они способны увеличиваться и уменьшаться по требованию. Наконец, подобно спискам словари – это гибкий инструмент для представления коллекций, но их *мнемонические* ключи лучше подходят, когда элементы коллекции именованы или помечены, скажем, как поля в записи базы данных.

Операции над отображениями

При написании в виде литералов словари указываются в фигурных скобках и состоят из ряда пар “ключ: значение”. Словари удобны всегда, когда нам необходимо ассоциировать набор значений с ключами – например, для описания свойств чего-нибудь. Рассмотрим следующий словарь их трех элементов (с ключами 'food', 'quantity' и 'color', возможно представляющих детали позиции гипотетического меню):

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

Мы можем индексировать этот словарь по ключу, чтобы извлекать и изменять значения, связанные с ключами. Операция индексации словаря имеет такой же синтаксис, как для последовательностей, но элементом в квадратных скобках будет ключ, а не относительная позиция:

```
>>> D['food']                  # Извлечь значение, связанное с ключом 'food'
'Spam'
>>> D['quantity'] += 1        # Добавить 1 к значению, связанному с ключом 'quantity'
>>> D
{'color': 'pink', 'food': 'Spam', 'quantity': 5}
```

Хотя форма литерала в фигурных скобках встречается, пожалуй, чаще приходится видеть словари, построенные другими способами (все данные программы редко известны до ее запуска). Скажем, следующий код начинает с пустого словаря и заполняет его по одному ключу за раз. В отличие от присваивания элементу в списке, находящемуся вне установленных границ, которое запрещено, присваивание новому ключу словаря приводит к созданию этого ключа:

```
>>> D = {}
>>> D['name'] = 'Bob'       # Присваивание приводит к созданию ключей
>>> D['job'] = 'dev'
>>> D['age'] = 40
```

```
>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}
>>> print(D['name'])
Bob
```

Здесь мы фактически применяем ключи словаря как имена полей в записи, представляющей кого-нибудь. В других приложениях словари могут использоваться также для замены операций поиска – индексирование словаря по ключу зачастую оказывается самым быстрым способом реализации поиска в Python.

Позже будет показано, что мы можем создавать словари также путем передачи имени типа `dict` либо *аргументов с ключевыми словами* (специальный синтаксис `имя=значение` в вызовах функций), либо результата *связывания* вместе последовательностей ключей и значений, полученных во время выполнения (например, из файлов). Оба следующих фрагмента создают тот же самый словарь, что и в предыдущем примере, и его эквивалентную литеральную форму {}, но второй фрагмент сопряжен с меньшим объемом ввода:

```
>>> bob1 = dict(name='Bob', job='dev', age=40)           # Ключевые слова
>>> bob1
{'age': 40, 'name': 'Bob', 'job': 'dev'}
>>> bob2 = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40]))
# Связывание вместе
>>> bob2
{'job': 'dev', 'name': 'Bob', 'age': 40}
```

Обратите внимание на то, как *перепутывается* порядок слева направо для ключей словаря. Отображения не являются позиционно упорядоченными, поэтому (если только вам не повезет) они возвращаются в порядке, отличающемся от того, в котором вы их набирали. Точный порядок может варьироваться в зависимости от версии Python, но вы не должны на него полагаться, как и не должны ожидать получения результатов, совпадающих с приведенными в книге.

Снова о вложении

В предыдущем примере мы использовали словарь с тремя ключами для описания гипотетического лица. Однако предположим, что информация оказывается сложнее. Пусть нам нужно хранить имя и фамилию и несколько названий должностей. В итоге мы имеем еще одно применение вложения объектов Python в действии. В приведенном далее словаре, одновременно представленном в виде литерала, содержится более структурированная информация:

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
          'jobs': ['dev', 'mgr'],
          'age': 40.5}
```

Мы снова имеем словарь с тремя ключами (`'name'`, `'jobs'` и `'age'`), но значения стали более сложными: вложенный словарь для имени, поддерживающий несколько частей, и вложенный список для названий должностей, поддерживающий много названий и будущее расширение. Мы можем получать доступ к компонентам этой структуры почти так же, как ранее к матрице, основанной на списках, но теперь большинство индексов являются ключами в словарях, а не смещениями в списках:

```
>>> rec['name']                                # 'name' - вложенный словарь
{'last': 'Smith', 'first': 'Bob'}
>>> rec['name']['last']                         # Индексация во вложенном словаре
'Smith'
```

```

>>> rec['jobs']
['dev', 'mgr']
>>> rec['jobs'][-1]
'mgr'
>>> rec['jobs'].append('janitor') # Расширение списка названий
# должностей на месте
>>> rec
{'age': 40.5, 'jobs': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith',
'first': 'Bob'}}

```

Обратите внимание на то, как последняя операция расширяет вложенный список названий должностей – поскольку список находится в области памяти, отдельной от словаря, который его содержит, он может свободно увеличиваться и уменьшаться (схема размещения объектов в памяти будет подробно обсуждаться позже в книге).

Настоящая причина, по которой показан этот пример, связана с демонстрацией гибкости основных типов данных Python. Вы можете заметить, что вложение позволяет строить сложные информационные структуры непосредственно и легко. Построение похожей структуры на низкоуровневом языке C было бы утомительным и требовало бы гораздо большего объема кода: пришлось бы планировать и объявлять структуры и массивы, заполнять их значениями, увязывать все вместе и т.д. В Python все происходит автоматически – выполнение выражения создает целую структуру вложенных объектов. На самом деле это одно из главных преимуществ языков написания сценариев, подобных Python.

Не менее важно и то, что в случае языка более низкого уровня мы должны обеспечить очистку области памяти, занимаемой всеми объектами, когда они больше не нужны. В случае Python, когда утрачивается последняя ссылка на объект (скажем, за счет присваивания его переменной чего-нибудь другого), то вся область памяти, занимаемая структурой этого объекта, очищается автоматически:

```
>>> rec = 0      # Теперь область памяти, занимаемая объектом, восстановлена
```

Говоря формально, в Python имеется средство, называемое *сборкой мусора*, которое очищает неиспользуемую память во время выполнения программы и освобождает вас от обязанности заниматься такими деталями в своем коде. В стандартном Python (CPython) область памяти восстанавливается немедленно после того, как удаляется последняя ссылка на объект. Мы исследуем работу сборки мусора в главе 6, а пока достаточно знать, что объекты можно свободно использовать, не беспокоясь о выделении им области памяти или ее очистке, когда необходимость в них исчезает.

Также наблюдайте за структурой записи, подобной той, что реализуется в главах 8, 9 и 27, которая будет применяться для сравнения и противопоставления списков, словарей, кортежей, именованных кортежей и классов – массив вариантов структур данных с компромиссами, полностью раскрываемый позже⁴.

⁴ Два замечания касательно применимости. Во-первых, только что созданная запись `rec` действительно может быть реальной записью базы данных, когда мы эксплуатируем систему *постоянства объектов Python* (легкий способ хранения собственных объектов Python в простых файлах или базах данных с доступом по ключу), которая автоматически транслирует объекты в и из байтовых потоков. Здесь мы не будем погружаться в детали, но дождитесь рассмотрения модулей Python для постоянства `pickle` и `shelve` в главах 9, 28, 31 и 37, где они исследуются в контексте файлов, сценария использования в объектно-ориентированном программировании, классов и изменений Python 3.X соответственно.

Недостающие ключи: проверки `if`

Как отображения, словари обеспечивают доступ к элементам только по ключу посредством только что показанных операций. Тем не менее, они дополнительно поддерживают операции, специфические для типа, через вызовы *методов*, которые полезны в разнообразных ситуациях. Например, несмотря на то, что мы можем выполнять присваивание новому ключу для расширения словаря, извлечение несуществующего ключа по-прежнему расценивается как ошибка:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> D['e'] = 99      # Присваивание новому ключу увеличивает словарь
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}
>>> D['f']          # Ссылка на несуществующий ключ приводит к ошибке
...текст сообщения об ошибке не показан...
KeyError: 'f'
```

Это именно то, что нужно – извлечение чего-то несуществующего обычно является программной ошибкой. Но в некоторых общих программах при написании кода мы не всегда можем знать, какие ключи будут присутствовать в словарях во время выполнения. Как обрабатывать такие сценарии, чтобы избежать ошибок? Одно из решений предусматривает заблаговременную проверку. Выражение проверки членства в словаре, `in`, позволяет запрашивать существование ключа и организовать ветвление в зависимости от результата с помощью Python-оператора `if`. Во время ввода следующего кода не забудьте нажать клавишу `<Enter>` два раза для интерактивного выполнения оператора `if` (как объяснялось в главе 3, пустая строка в интерактивной подсказке означает “выполнить”). Кроме того, как и при вводе примеров многострочных словарей и списков ранее в главе, в некоторых интерфейсах приглашение изменяется на ... для второй строки и далее:

```
>>> 'f' in D
False
>>> if not 'f' in D:      # Единственный оператор выбора Python
    print('missing')
missing
```

Оператор `if` еще будет обсуждаться в более поздних главах книги, но применяемая здесь форма прямолинейна: она состоит из слова `if`, за которым следует выражение, дающее истинный или ложный результат, и блок кода, подлежащий выполнению в случае истинного результата проверки.

Во-вторых, если вы знакомы с *JSON* (*JavaScript Object Notation* – система обозначений для объектов *JavaScript*), который представляет собой развивающийся формат обмена данными, применяемый для баз данных и передачи по сети, тогда этот пример также может выглядеть на удивление похожим, хотя поддержка переменных, произвольные выражения и изменения Python могут сделать его структуры данных более универсальными. Библиотечный модуль Python по имени `json` поддерживает создание и разбор текста JSON, но трансляция в объекты Python часто тривиальна. Созданная запись `rec` будет задействована в примере использования JSON в главе 9 при изучении файлов. За более крупным сценарием применения обратитесь к системе управления базами данных *MongoDB*, которая хранит данные с использованием нейтральной к языкам двоичной сериализации JSON-подобных документов, и ее интерфейсу *PyMongo*.

В своей полной форме оператор `if` может также иметь конструкцию `else` для стандартного случая и одну или более конструкций `elif` ("else if") для других проверок. Он является главным инструментом *выбора* в Python; наряду с родственным ему тернарным выражением `if/else` (которое вскоре будет описано) и похожим на него фильтром `if` списковых включений, который был показан ранее, он представляет собой способ написания логики выбора и принятия решений в сценариях.

Если вы использовали другие языки программирования в прошлом, то вас может интересовать, каким образом Python узнает, где заканчивается оператор `if`. Правила синтаксиса Python будут объясняться позже в книге, но вкратце, если в блоке операторов должно выполняться более одного действия, тогда нужно просто обеспечить всем им одинаковые отступы. Такой подход соответствует читабельности кода и сокращает объем ввода:

```
>>> if not 'f' in D:  
    print('missing')  
    print('no, really...')    # Блоки операторов вводятся с отступом  
missing  
no, really...
```

Кроме проверки посредством `in` существует еще много других способов избежать обращения к несуществующим ключам в создаваемых словарях: метод `get` (условный индекс со стандартным вариантом); метод `has_key` из Python 2.X (аналог `in`, недоступный в Python 3.X); оператор `try` (рассматриваемый в главе 10 инструмент, который обеспечивает перехват и восстановление после исключений); и тернарное (состоящее из трех частей) выражение `if/else`, которое по существу является оператором `if`, сжатым в одну строку. Вот несколько примеров:

```
>>> value = D.get('x', 0)                      # Индекс со стандартным вариантом  
>>> value  
0  
>>> value = D['x'] if 'x' in D else 0      # Форма выражения if/else  
>>> value  
0
```

Более подробно указанные альтернативы обсуждаются позже в книге. Теперь давайте перейдем к исследованию роли еще одного метода словаря в распространенном сценарии применения.

Сортировка ключей: циклы `for`

Как упоминалось ранее, поскольку словари – не последовательности, они не поддерживают сколько-нибудь надежный порядок слева направо. Если создать словарь и сразу его вывести, то ключи могут следовать не в том порядке, в котором они набирались, и варьироваться в зависимости от версии Python и других переменных:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}  
>>> D  
{'a': 1, 'c': 3, 'b': 2}
```

Однако что делать, если необходимо навязать какой-то порядок элементам словаря? Распространенное решение предусматривает получение списка ключей с помощью метода `keys` словаря, сортировку этого списка посредством спискового метода `sort` и проход по результату в цикле `for` языка Python (как и в случае `if`, после ввода следующего цикла `for` обязательно нажмите клавишу `<Enter>` два раза и удалите внешние круглые скобки в `print`, если работаете в Python 2.X):

```

>>> Ks = list(D.keys())
# Неупорядоченный список ключей
>>> Ks
# Список в Python 2.X, "представление"
# в Python 3.X: вызовите list()
['a', 'c', 'b']

>>> Ks.sort()
# Отсортированный список ключей

>>> Ks
['a', 'b', 'c']

>>> for key in Ks:           # Проход по отсортированным ключам
    print(key, '=>', D[key]) # <== здесь нажмите <Enter>
                                # два раза (print версии Python 3.X)

a => 1
b => 2
c => 3

```

Процесс состоит из трех шагов, хотя, как вы увидите далее в книге, в последних версиях Python задачу можно решить за один шаг посредством новой встроенной функции `sorted`. Вызов `sorted` возвращает результат и сортирует объекты разнообразных типов, в данном случае сортируя ключи словаря автоматически:

```

>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> for key in sorted(D):
    print(key, '=>', D[key])

a => 1
b => 2
c => 3

```

Помимо демонстрации работы со словарями приведенный сценарий использования служит введением в цикл `for` языка Python. Цикл `for` является простым и эффективным способом прохода по всем элементам в последовательности и выполнения блока кода для каждого элемента по очереди. Для ссылки на текущий элемент применяется определяемая пользователем переменная цикла (здесь `key`). Совокупным эффектом примера будет вывод ключей и значений неупорядоченного словаря в порядке сортировки ключей.

Цикл `for` и более универсальный цикл `while` предоставляют главные способы написания кода для выполнения *повторяющихся* задач в виде операторов в сценариях. Тем не менее, в действительности цикл `for`, как и родственное ему списковое включение, которое рассматривалось ранее, является операцией над последовательностью. Он работает на любом объекте, относящемся к последовательностям, и подобно списковому включению даже на некоторых вещах, не считающихся последовательностями. Например, вот как можно пройти по символам в строке с выводом их версии в верхнем регистре:

```

>>> for c in 'spam':
    print(c.upper())

S
P
A
M

```

Цикл `while` языка Python представляет собой более универсальную разновидность инструментов для организации циклов; он не ограничивается проходом через последовательности, но в целом требует написания большего объема кода:

```
>>> x = 4
>>> while x > 0:
...     print('spam!' * x)
...     x -= 1
...
spam!spam!spam!
spam!spam!spam!
spam!spam!
spam!
```

Мы подробно обсудим операторы циклов, синтаксис и инструменты позже в книге. Но сначала мне нужно признаться, что в настоящем разделе я не был настолько откровенен, насколько мог бы быть. По правде говоря, цикл `for` и все его категории, выполняющие проход по объектам слева направо, являются не просто операциями над *последовательностями*, а операциями над *итерируемыми* объектами, как объясняется в следующем разделе.

Итерация и оптимизация

Если цикл `for` в предыдущем разделе выглядит подобно выражению спискового включения, введенному ранее, то так и должно быть: на самом деле они оба представляют собой универсальные инструменты для итерации. Фактически они оба будут работать на любом итерируемом объекте, который следует протоколу итерации — всемпроникающей идеи в Python, лежащей в основе всех инструментов для итерации.

Говоря кратко, объект считается *итерируемым*, если он является либо физически сохраненной последовательностью в памяти, либо объектом, который генерирует под одному элементу за раз в контексте итерационной операции — своего рода “виртуальной” последовательностью. Более формально оба типа объектов считаются итерируемыми, поскольку поддерживают *протокол итерации* — они отвечают на вызов `iter` значением, которое продвигается вперед в ответ на вызовы `next`, и генерируют исключение, когда завершают выпуск значений.

К таким объектам относится выражение спискового включения для *генератора*, которое обсуждалось ранее: его значения вообще не хранятся в памяти, а предоставляются по запросу, обычно выполняемому инструментами итерации. *Объекты файлов* Python аналогичным образом производят итерации по строкам, когда используются инструментом итерации: содержимое файла — не список, оно извлекается по требованию. Оба являются итерируемыми объектами в Python, относясь к категории, которая в версии Python 3.X была расширена, чтобы включать основные инструменты вроде `range` и `map`. Откладывая получение результатов по мере необходимости, эти инструменты могут экономить память и минимизировать задержки.

Мы рассмотрим протокол итерации более подробно далее в книге. А пока запомните, что любой инструмент Python, который проходит по объекту слева направо, действует согласно протоколу итерации. Именно потому вызов `sorted`, применяемый в предыдущем разделе, работал напрямую на словаре — мы не обязаны вызывать метод `keys` для получения последовательности, т.к. словари являются итерируемыми объектами с методом `next`, который возвращает последовательные ключи.

Помочь пониманию способен тот факт, что любое выражение спискового включения, например вычисляющее квадраты для списка чисел:

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
>>> squares
[1, 4, 9, 16, 25]
```

всегда можно переписать в виде эквивалентного цикла `for`, который строит результатирующий список вручную, дополняя его на каждом шаге:

```
>>> squares = []
>>> for x in [1, 2, 3, 4, 5]:
...     squares.append(x ** 2)
... # Это то, что делает списковое включение
... # Оба инструмента внутренне следуют
... # протоколу итерации

>>> squares
[1, 4, 9, 16, 25]
```

Оба инструмента внутренне задействуют протокол итерации и производят тот же самый результат. Однако списковое включение и связанные инструменты функционального программирования вроде `map` и `filter` в наши дни часто будут выполняться быстрее цикла `for` в коде определенных видов (возможно даже в два раза) – характеристика, которая может быть значимой в программах, обрабатывающих крупные наборы данных. Но после сказанного я должен отметить, что показатели производительности в языке Python – вещи не особо надежные, потому что он оптимизирует слишком многое, и они могут меняться от выпуска к выпуску.

Основное эмпирическое правило в Python таково: писать код, заботясь в первую очередь о простоте и читабельности, а о производительности беспокоиться позже, когда программа заработала, и после того, как вы докажете наличие подлинной проблемы с производительностью. Почти всегда ваш код будет достаточно быстрым в том виде, как есть. Тем не менее, если вам нужно подстроить код ради достижения более высокой производительности, то в состав Python входят модули `time` и `timeit` для измерения скорости альтернатив и модуль `profile` для устранения узких мест.

Более подробно о таких темах вы узнаете позже в книге (особенно в учебном примере оценочного испытания из главы 21) и в руководствах Python. А сейчас мы переходим к следующему основному типу данных.

Кортежи

Объект кортежа примерно похож на список, который нельзя изменять – кортежи являются *последовательностями* подобно спискам, но они *неизменяемые* подобно строками. Функционально они используются для представления фиксированных коллекций элементов: скажем, компонентов специфической даты в календаре. Синтаксически они записываются в круглых, а не квадратных скобках и поддерживают произвольные типы, произвольное вложение и обычные операции над последовательностями:

```
>>> T = (1, 2, 3, 4)      # Кортеж из 4 элементов
>>> len(T)                # Длина
4
>>> T + (5, 6)            # Конкатенация
(1, 2, 3, 4, 5, 6)
>>> T[0]                  # Индексация, нарезание и т.д.
1
```

Кортежи в Python 2.6 и 3.0 также имеют вызываемые методы, специфичные для типа, но далеко не так много, как списки:

```
>>> T.index(4)            # Методы кортежей: 4 обнаруживается по смещению 3
3
>>> T.count(4)             # 4 обнаруживается один раз
1
```

Главное отличие кортежей заключается в том, что после создания их нельзя изменять, т.е. они являются неизменяемыми последовательностями (одноэлементные кортежи вроде приведенного ниже требуют хвостовой запятой):

```
>>> T[0] = 2          # Кортежи неизменяемы
... текст сообщения об ошибке не показан...
TypeError: 'tuple' object does not support item assignment
Ошибка типа: объект tuple не поддерживает присваивание в отношении элементов
>>> T = (2,) + T[1:]    # Создает новый кортеж для нового значения
>>> T
(2, 2, 3, 4)
```

Подобно спискам и словарям кортежи поддерживают смешанные типы и вложение, но не увеличиваются и не уменьшаются, поскольку они неизменяемы (круглые скобки, окружающие элементы кортежа, часто можно опускать, как сделано здесь; запятые – это то, что фактически строит кортеж):

```
>>> T = 'spam', 3.0, [11, 22, 33]
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
Ошибка атрибута: объект tuple не имеет атрибута append
```

Для чего используются кортежи?

Итак, зачем нам тип, который похож на список, но поддерживает меньше операций? Откровенно говоря, на практике кортежи применяются в целом не так часто, как списки, но весь смысл в их неизменяемости. Если вы передаете коллекцию объектов внутри своей программы в виде списка, тогда он может быть модифицирован где угодно; если вы используете кортеж, то изменить его не удастся. То есть кортежи обеспечивают своего рода ограничение целостности, что удобно в программах, крупнее тех, которые мы будем писать здесь. Позже в книге мы еще обсудим кортежи, включая расширение, которое построено поверх них и называется *именованными кортежами*. Теперь давайте займемся последним из важных основных типов: файлом.

Файлы

Объекты файлов являются главным интерфейсом к внешним файлам на компьютере. Они могут применяться для чтения и записи текстовых заметок, аудиоклипов, документов Excel, сохраненных сообщений электронной почты и всего того, что вы в итоге сохранили на своем компьютере. Файлы относятся к основным типам, но они кое в чем своеобразны – специфический литературный синтаксис для их создания отсутствует. Взамен, чтобы создать объект файла, необходимо вызвать встроенную функцию open, передав ей в виде строк имя внешнего файла и необязательный режим обработки.

Например, для создания выходного текстового файла понадобится передать его имя и строку режима обработки 'w', чтобы записывать данные:

```
>>> f = open('data.txt', 'w') # Создать новый файл в режиме записи ('w')
>>> f.write('Hello\n')        # Записать в него строки символов
```

```
6  
>>> f.write('world\n')          # Возвратить количество записанных элементов  
                                # в версии Python 3.X  
6  
>>> f.close()                 # Закрыть для сбросывания буферов вывода на диск
```

Код создает файл в текущем каталоге и записывает в него текст (имя файла может содержать полный путь к каталогу, если нужно получить доступ к файлу где-то в другом месте на компьютере). Чтобы прочитать то, что было записано, необходимо повторно открыть файл в режиме обработки 'r' для чтения текстового ввода (он выбирается по умолчанию, если в вызове строка режима не указана). Затем следует прочитать содержимое файла в строку и отобразить ее. В сценарии содержимое файла всегда будет строкой независимо от типа находящихся в нем данных:

```
>>> f = open('data.txt')         # 'r' (чтение) - стандартный режим обработки  
>>> text = f.read()            # Прочитать все содержимое файла в строку  
>>> text  
'Hello\nworld\n'  
  
>>> print(text)                # print интерпретирует управляемые символы  
Hello  
world  
  
>>> text.split()               # Содержимое файла - всегда строка  
['Hello', 'world']
```

Другие методы объектов файлов поддерживают дополнительные возможности, которые здесь раскрываться не будут. Например, объекты файлов предоставляют больше способов чтения и записи (`read` принимает необязательный максимальный размер в байтах/символах, `readline` читает по одной строке за раз и т.д.), а также другие инструменты (`seek` перемещает в новую позицию внутри файла). Однако как вы увидите позже, лучший способ чтения файла в наше время — *вообще его не читать*, поскольку файлы предлагают *итератор*, который автоматически производит чтение строки за строкой в циклах `for` и других контекстах:

```
>>> for line in open('data.txt'): print(line)
```

Мы будем сталкиваться с полным набором методов файлов в последующих главах, но если вы хотите получить краткий обзор прямо сейчас, тогда вызовите `dir` для любого открытого файла и `help` для любого имени метода, выведенного `dir`:

```
>>> dir(f)  
[...много имен не показано...  
'buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno', 'flush',  
'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable',  
'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate',  
'writable',  
'write', 'writelines']  
  
>>> help(f.seek)  
...попробуйте и просмотрите...
```

Файлы с двоичными байтами

Примеры в предыдущем разделе иллюстрировали основы, которых во многих ситуациях вполне достаточно. Тем не менее, формально они полагаются либо на кодировку Unicode платформы, стандартную в Python 3.X, либо на 8-битную природу байтов в файлах Python 2.X. Текстовые файлы всегда кодируют строки в Python 3.X и слепо

записывают содержимое строк в Python 2.X. Это не относится к используемым ранее простым данным ASCII, которые отображаются на байты в файле без изменений. Но для более развитых типов данных интерфейсы к файлам могут варьироваться в зависимости от содержимого и применяемой линейки Python.

Ранее при обсуждении строк было указано, что версия Python 3.X проводит четкое различие между текстовыми и двоичными данными в файлах. *Текстовые файлы* представляют содержимое в виде нормальных строк `str`, автоматически выполняя кодирование и декодирование Unicode при записывании и чтении данных, тогда как *двоичные файлы* представляют содержимое в виде специальных строк `bytes` и позволяют получать доступ к содержимому файла, не изменяя его. В Python 2.X поддерживается та же самая двойственность, но она не навязывается настолько жестко, и инструменты отличаются.

Скажем, *двоичные файлы* удобны для обработки медиаданных, доступа к данным, созданным программами С и т.д. В качестве иллюстрации Python-модуль `struct` способен создавать и распаковывать упакованные *двоичные данные* (неформатированные байты, фиксирующие значения, которые не являются объектами Python), подлежащие записи в двоичном режиме. Мы изучим эту методику позже в книге, но концепция проста: следующий код создает двоичный файл в Python 3.X (в Python 2.X двоичные файлы работают аналогично, но префикс `b` строкового литерала не требуется и не отображается):

```
>>> import struct
>>> packed = struct.pack('>i4sh', 7, b'spam', 8) # Создать упакованные
                                                # двоичные данные
>>> packed                                     # 10 байтов, не объекты и не текст
b'\x00\x00\x00\x07spam\x00\x08'
>>>
>>> file = open('data.bin', 'wb')   # Открыть двоичный файл для записи
>>> file.write(packed)           # Записать упакованные двоичные данные
10
>>> file.close()
```

Чтение двоичных данных по существу симметрично; не всем программам нужно настолько глубоко проникать в низкоуровневую область байтов, но двоичные файлы в Python облегчают задачу:

```
>>> data = open('data.bin', 'rb').read()      # Открыть/прочитать
                                                # двоичный файл данных
                                                # 10 байтов, неизмененные
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]                                # Нарезать байты в середине
b'spam'
>>> list(data)                            # Последовательность 8-битных байтов
[0, 0, 0, 7, 115, 112, 97, 109, 0, 8]
>>> struct.unpack('>i4sh', data)          # Снова распаковать в объекты
(7, b'spam', 8)
```

Файлы с текстом Unicode

Текстовые файлы используются для обработки всевозможных текстовых данных, начиная с заметок и содержимого сообщений электронной почты и заканчивая документами JSON и XML. Но в современном более широко взаимосвязанном мире мы не можем говорить о тексте, не задавая вопрос о виде текста. Вы должны знать тип кодировки Unicode текста, если он отличается от принятого по умолчанию на вашей

платформе или вы не можете полагаться на стандартную кодировку по причинам, связанным с переносимостью данных.

К счастью, все проще, чем могло показаться. Чтобы получить доступ к файлам, содержащим отличающийся от ASCII текст *Unicode* представленного ранее вида, мы просто передаем имя кодировки, если она не совпадает со стандартной кодировкой для имеющейся платформы. В таком режиме текстовые файлы Python автоматически *кодируют* при записи и *декодируют* при чтении согласно схеме кодировки, имя которой было предоставлено. В *Python 3.X*:

```
>>> S = 'sp\xc4m'           # Текст Unicode, отличающийся от ASCII
>>> S
'spAm'
>>> S[2]                  # Последовательность символов
'A'
>>> file = open('unidata.txt', 'w', encoding='utf-8') # Записать/
                                                # закодировать текст UTF-8
>>> file.write(S)          # Записано 4 символа
4
>>> file.close()
>>> text = open('unidata.txt', encoding='utf-8').read() # Прочитать/
                                                # декодировать текст UTF-8
>>> text
'spAm'
>>> len(text)              # 4 символа (кодовые точки)
4
```

Такое автоматическое кодирование и декодирование является тем, что обычно требуется. Поскольку файлы делают это при передаче, вы можете обрабатывать текст в памяти как простую строку символов, не заботясь о его источниках в кодировке Unicode. Однако при необходимости вы можете посмотреть, что действительно хранится в файле, перейдя в двоичный режим:

```
>>> raw = open('unidata.txt', 'rb').read()    # Читать закодированные байты
>>> raw
b'sp\xc3\x84m'
>>> len(raw)                      # 5 байтов в кодировке UTF-8
5
```

В случае, когда данные Unicode получаются не из файла (скажем, из сообщения электронной почты или через сетевое подключение), вы также можете их кодировать и декодировать вручную:

```
>>> text.encode('utf-8')      # Вручную кодировать в байты
b'sp\xc3\x84m'
>>> raw.decode('utf-8')       # Вручную декодировать в строку
'spAm'
```

Прием удобен, если нужно посмотреть, как текстовые файлы автоматически кодировали бы ту же самую строку для разных имен кодировок, и он представляет собой способ перевода данных в другие кодировки. В случае указания надлежащего имени кодировки отличающиеся байты в файлах декодируются в ту же самую строку в памяти:

```
>>> text.encode('latin-1')    # Байты отличаются от других
b'sp\xc4m'
>>> text.encode('utf-16')
b'\xff\xfe\x00p\x00\xc4\x00m\x00'
```

```
>>> len(text.encode('latin-1')), len(text.encode('utf-16'))
(4, 10)
>>> b'\xff\xfe\x00p\x00\xc4\x00m\x00'.decode('utf-16') # Но декодируются
# в ту же самую строку
'spAm'
```

В версии *Python 2.X* все работает более или менее одинаково, но строки Unicode вводятся и отображаются с ведущим символом `u`, байтовые строки не требуют ведущего символа `b`, текстовые файлы Unicode должны открываться с помощью функции `codecs.open`, которая подобно `open` из *Python 3.X* принимает имя кодировки, и для представления содержимого в памяти применяется специальная строка `unicode`. Режим двоичного файла в *Python 2.X* может показаться необязательными, т.к. обычные файлы представляют собой данные, основанные на байтах, но он требуется для того, чтобы избежать изменения символов конца строки, если они присутствуют:

```
>>> import codecs
>>> codecs.open('unidata.txt', encoding='utf8').read() # Python 2.X: читать/
# декодировать текст
u'sp\xc4m'
>>> open('unidata.txt', 'rb').read() # Python 2.X: читать низкоуровневые байты
'sp\xc3\x84m'
>>> open('unidata.txt').read() # Python 2.X: также низкоуровневые/декодированные
'sp\xc3\x84m'
```

Хотя заботиться о таком отличии обычно не приходится, если вы имеете дело только с текстом ASCII, строки и файлы являются ценным ресурсом в случае работы с двоичными данными (которые охватывают большинство видов медиаданных) или текстом с интернационализированными наборами символов (который включает большинство содержимого веб-сети и Интернета в целом). Язык *Python* также поддерживает имена файлов, отличающиеся от ASCII (не только содержимое), но это делается в значительной степени автоматически, как будет показано в главе 37.

Другие инструменты, подобные файлам

Функция `open` является основополагающим компонентом при выполнении большинства работ с файлами в *Python*. Тем не менее, в составе *Python* имеются дополнительные инструменты для решения более сложных задач: конвейеры, очереди FIFO, сокеты, файлы с доступом по ключу, модуль `shelve` для постоянства объектов, дескрипторные файлы, интерфейсы к реляционным и объектно-ориентированным базам данных и многое другое. Например, дескрипторные файлы поддерживают блокировку и прочие низкоуровневые инструменты, а сокеты предоставляют интерфейс для работы в сети и взаимодействия между процессами. Многие из перечисленных инструментов в настоящей книге не рассматриваются, но вы сочтете их полезными, когда всерьез начнете программировать на *Python*.

Прочие основные типы

Помимо основных типов, описанных до сих пор, существуют и другие, которые могут или не могут претендовать на членство в категории в зависимости от того, насколько широко она определена. Скажем, множества представляют собой недавнее добавление к языку и не являются ни отображениями, ни последовательностями; на самом деле это неупорядоченные коллекции уникальных и неизменяемых объектов. Множества создаются путем вызова встроенной функции `set` либо использования

новых литералов и выражений множеств в Python 3.X и 2.7 и поддерживают обычные математические операции над множествами (выбор нового синтаксиса `{...}` для литералов множеств логичен, т.к. множества очень похожи на ключи в словарях без значений):

```
>>> X = set('spam')          # Создать множество из последовательности
                                # в Python 2.X и 3.X
>>> Y = {'h', 'a', 'm'}      # Создать множество с помощью литералов
                                # множеств в Python 3.X и 2.7
>>> X, Y                   # Кортеж из двух множеств без круглых скобок
({'m', 'a', 'p', 's'}, {'m', 'a', 'h'})
>>> X & Y                 # Пересечение
{'m', 'a'}
>>> X | Y                 # Объединение
{'m', 'h', 'a', 'p', 's'}
>>> X - Y                 # Разность
{'p', 's'}
>>> X > Y                 # Надмножество
False
>>> {n ** 2 for n in [1, 2, 3, 4]} # Включения множеств в Python 3.X и 2.7
{16, 1, 4, 9}
```

Даже не особо математически подкованные программисты нередко находят множества удобными для решения общих задач, таких как фильтрация дубликатов, нахождение разностей и выполнение нейтральных к порядку проверок равенства без сортировки в списках, строках и всех остальных итерируемых объектах:

```
>>> list(set([1, 2, 1, 3, 1]))    # Фильтрация дубликатов
                                # (возможно неупорядоченных)
[1, 2, 3]
>>> set('spam') - set('ham')      # Нахождение разностей в коллекциях
{'p', 's'}
>>> set('spam') == set('asmp')    # Нейтральная к порядку проверка
                                # равенства ('spam'=='asmp' дает False)
True
```

Множества также поддерживают проверки членства `in`, хотя все остальные типы коллекций Python тоже это делают:

```
>>> 'p' in set('spam'), 'p' in 'spam', 'ham' in ['eggs', 'spam', 'ham']
(True, True, True)
```

Вдобавок в Python недавно появилось несколько новых числовых типов: *десятичные* числа, которые представляют собой числа с плавающей точкой фиксированной точности, и *дробные* числа, которые являются рациональными числами с числителем и знаменателем. Оба типа могут применяться для обхода ограничений и свойственной неточности математики с плавающей точкой:

```
>>> 1 / 3          # Математика с плавающей точкой (добавьте .0 в Python 2.X)
0.3333333333333333
>>> (2/3) + (1/2)
1.1666666666666665
>>> import decimal # Десятичные числа: фиксированная точность
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal('4.141')
```

```

>>> decimal.getcontext().prec = 2
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33')
>>> from fractions import Fraction    # Дроби: числитель + знаменатель
>>> f = Fraction(2, 3)
>>> f + 1
Fraction(5, 3)
>>> f + Fraction(1, 2)
Fraction(7, 6)

```

В Python также имеются *булевские* значения (с предопределенными объектами `True` и `False`, которые по существу представляют собой целые числа 1 и 0 со специальной логикой отображения) и долго поддерживаемый особый объект-заполнитель по имени `None`, часто используемый для инициализации имен и объектов:

```

>>> 1 > 2, 1 < 2          # Булевские значения
(False, True)
>>> bool('spam')         # Булевское значение объекта
True
>>> X = None             # Заполнитель None
>>> print(X)
None
>>> L = [None] * 100      # Инициализировать список сотней объектов None
>>> L
[None, None, None,
None, None, None, None, None, None, None, ...список из 100 объектов None...]

```

Как нарушить гибкость кода

Позже вы еще многое узнаете обо всех типах объектов Python, но один из них заслуживает особого внимания. Объект `type`, возвращаемый встроенной функцией `type`, является объектом, который дает тип другого объекта; результат функции `type` в Python 3.X слегка отличается, поскольку типы были полностью объединены с классами (то, что мы будем исследовать в контексте классов “нового стиля” в части VI). Предположим, что `L` – все еще список из предыдущего раздела:

```

# В Python 2.X:
>>> type(L)                  # Типы: типом L является list
<type 'list'>
>>> type(type(L))           # Даже типы являются объектами
<type 'type'>
# В Python 3.X:
>>> type(L)                  # Python 3.X: типы являются классами и наоборот
<class 'list'>
>>> type(type(L))           # Типы классов более подробно рассматриваются в главе 32
<class 'type'>

```

Кроме возможности интерактивного исследования ваших объектов объект `type` в большинстве практических приложений позволяет коду проверять типы обрабатываемых объектов. В сущности, делать это в сценарии Python можно, по меньшей мере, тремя способами:

```

>>> if type(L) == type([]):    # Проверка типа при необходимости...
    print('yes')
yes
>>> if type(L) == list:       # Использование имени типа
    print('yes')

```

```
yes
>>> if isinstance(L, list):      # Объектно-ориентированная проверка
    print('yes')
yes
```

Теперь, когда вы увидели все способы выполнения проверки типов, я обязан сообщить вам, что поступать так в программе Python почти всегда неправильно (и это часто служит признаком того, что бывший программист на С впервые начал использовать Python). Причина станет полностью ясной лишь позже в книге, когда мы приступим к написанию более крупных единиц кода, таких как функции, но это является (*наверное*) основной концепцией Python. Выполнняя проверку на предмет специфического типа в своем коде, вы фактически нарушаете гибкость кода — ограничиваете его работой только с одним типом. Без такой проверки код может быть способен работать с целым диапазоном типов.

Все связано с упоминаемой ранее идеей *полиморфизма* и уходит своими корнями в отсутствие в Python объявлений типов. Как вы узнаете, в Python мы имеем дело с *интерфейсами* объектов (поддерживаемыми операциями), а не с типами. Другими словами нас заботит, что объект *делает*, а не чем он *является*. Отсутствие заботы о специфических типах означает, что код автоматически применим ко многим из них — будет работать любой объект с совместимым интерфейсом независимо от его конкретного типа. Хотя проверка типов поддерживается (и в редких случаях даже требуется), вы увидите, что она обычно отличается от обычного образа мышления Python. На самом деле вы обнаружите, что полиморфизм — вероятно, ключевая идея правильного использования Python.

Классы, определяемые пользователем

Объектно-ориентированное программирование на Python — необязательное, но мощное средство языка, которое сокращает время разработки за счет поддержки программирования через настройку, — подробно рассматривается позже в книге. В общих понятиях классы определяют новые типы объектов, которые расширяют основной набор, поэтому пока они заслуживают лишь мимолетного взгляда. Пусть, например, вы хотите иметь тип объекта, моделирующий сотрудника. Несмотря на отсутствие такого типа в наборе основных типов Python, следующий класс, определяемый пользователем, способен удовлетворить всем требованиям:

```
>>> class Worker:
    def __init__(self, name, pay):
        self.name = name                  # Инициализировать при создании
        self.pay = pay                      # self - новый объект
    def lastName(self):
        return self.name.split()[-1]       # Разбить строку по пробелам
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)        # Обновить pay на месте
```

Класс *Worker* определяет новый вид объекта, который будет иметь атрибуты *name* и *pay* (иногда называемые *информацией о состоянии*), а также две линии поведения, реализованные как функции (обычно называемые *методами*). Обращение к классу аналогично функции создает экземпляры нового типа, и методы класса автоматически получают экземпляр, обрабатываемый заданным вызовом метода (в аргументе *self*):

```
>>> bob = Worker('Bob Smith', 50000)    # Создать два экземпляра
>>> sue = Worker('Sue Jones', 60000)     # Каждый имеет атрибуты name и pay
```

```
>>> bob.lastName()                                # Вызвать метод: bob - это self
'Smith'
>>> sue.lastName()                               # sue - это self
'Jones'
>>> sue.giveRaise(.10)                            # Обновить атрибут pay в экземпляре sue
>>> sue.pay
66000.0
```

Подразумеваемый объект `self` является причиной названия такой модели *объектно-ориентированной*: в функциях внутри класса всегда присутствует подразумеваемый объект. Однако в некотором смысле тип, базирующийся на классе, просто построен и использует основные типы – например, определяемый пользователем объект `Worker` представляет собой всего лишь совокупность строки и числа (`name` и `pay` соответственно) плюс функции для обработки этих встроенных объектов.

Следует отметить, что механизм наследования классов поддерживает программные иерархии, которые поддаются настройке *расширением*. Мы расширяем программное обеспечение за счет написания новых классов, а не путем изменения того, что уже работает. Вы также должны знать, что классы являются необязательным средством Python, и более простые встроенные типы вроде списков и словарей часто оказываются лучшими инструментами, чем классы, определяемые пользователем. Тем не менее, все это выходит далеко за рамки вводного руководства по типам объектов, так что считайте изложенное просто обзором; полное раскрытие определяемых пользователем классов ищите далее в книге. Поскольку классы построены на основе других инструментов в Python, они входят в перечень главных целей настоящей книги.

Все остальное

Как упоминалось ранее, абсолютно все, что можно обрабатывать в сценарии Python, является какой-нибудь разновидностью объекта, поэтому наш экскурс в типы объектов далек от полноты. Однако хотя все в Python относится к объектам, только те типы объектов, которые мы встречали до сих пор, считаются частью набора основных типов Python. Остальные типы в Python будут либо объектами, связанными с выполнением программ (наподобие функций, модулей, классов и скомпилированного кода), которые мы исследуем позже, либо реализуются функциями импортированных модулей, а не синтаксисом языка. Последние также имеют тенденцию исполнять роли, специфичные для приложения – образцы текста, интерфейсы к базам данных, сетевые подключения и т.д.

Кроме того, имейте в виду, что рассмотренные до сих пор объекты являются объектами, но не обязательно в *объектно-ориентированном* смысле – концепция, которая обычно требует наследования и оператора `class` языка Python; далее в книге мы еще вернемся к ней. Тем не менее, основные объекты Python – это основополагающие компоненты почти каждого сценария Python, с которым вы столкнетесь, и они обычно выступают в качестве фундамента для более крупных неосновных типов.

Резюме

На этом наш начальный экскурс в типы данных завершен. В главе было предложено краткое введение в основные типы объектов Python и виды операций, которые можно к ним применять. Мы обсудили обобщенные операции, которые работают на многих типах объектов (например, операции над последовательностями, такие как индексация и нарезание), а также операции, специфичные для типов, которые до-

ступны как вызовы методов (скажем, разделение строки и добавление к списку). Мы определили ряд ключевых терминов, таких как неизменяемость, последовательности и полиморфизм.

Попутно вы увидели, что основные типы объектов Python являются более гибкими и мощными, чем все доступное в языках низкого уровня вроде С. Например, списки и словари Python избавляют от большинства работы, которую пришлось бы делать для поддержки коллекций и поиска в языках низкого уровня. Списки представляют собой упорядоченные коллекции других объектов, а словари – коллекции других объектов, которые индексируются по ключу вместо позиции. Словари и списки могут быть вложенными, увеличиваться и уменьшаться по требованию и содержать объекты любого типа. Кроме того, занимаемые ими области памяти автоматически очищаются, когда они больше не нужны. Также было показано, что строки и файлы работают рука об руку, чтобы поддерживать широкое многообразие двоичных и текстовых данных.

Для обеспечения краткого экскурса большинство деталей здесь было опущено, поэтому вы не должны ожидать, что все в главе станет понятным сразу. В последующих нескольких главах мы начнем погружаться глубже, совершая второй проход по основным типам объектов Python, который заполнит недостающие здесь детали и даст вам лучшее понимание. В следующей главе будут подробно рассматриваться числа Python. Но первым делом ответьте на традиционные контрольные вопросы.

Проверьте свои знания: контрольные вопросы

Представленные в главе концепции будут детально исследоваться в последующих главах, поэтому просто раскрывайте здесь главные идеи.

1. Назовите четыре основных типа данных Python.
2. Почему они называются “основными” типами данных?
3. Что означает “неизменяемость”, и какие три основных типа Python считаются неизменяемыми?
4. Что означает “последовательность”, и какие три типа входят в эту категорию?
5. Что означает “отображение”, и какой основной тип является отображением?
6. Что такое “полиморфизм” и почему о нем нужно заботиться?

Проверьте свои знания: ответы

1. В общем случае основными типами объектов (типами данных) считаются числа, строки, списки, словари, кортежи, файлы и множества. Сами типы, None и булевские значения иногда также классифицируются подобным образом. Существует много числовых типов (целые, с плавающей точкой, комплексные, дробные и десятичные) и несколько строковых типов (простые строки и строки Unicode в Python 2.X, а также текстовые строки и байтовые строки в Python 3.X).
2. Они также известны как “основные” типы, потому что являются частью самого языка Python и всегда доступны; чтобы создавать остальные объекты, обычно требуется вызов функций из импортированных модулей. Большинство основных типов имеют специфический синтаксис для создания объектов: например, 'spam' представляет собой выражение, создающее строку и определяющее набор операций, которые могут к ней применяться. Из-за этого основные типы

вщиты в синтаксис Python. Напротив, для создания объекта файла необходимо вызывать встроенную функцию open (хотя его обычно относят к основным типам).

3. “Неизменяемый” объект – это такой объект, который после создания нельзя модифицировать. В эту категорию входят числа, строки и кортежи Python. Наряду с тем, что неизменяемый объект нельзя изменять на месте, всегда можно создать новый объект, выполнив выражение. Байтовые массивы в недавних версиях Python предлагают неизменяемость для текста, но они не являются нормальными строками и применяются напрямую к тексту, только если он представляет собой простой 8-битный вид (например, ASCII).
4. “Последовательность” – это позиционно упорядоченная коллекция объектов. Последовательностями в Python являются строки, списки и кортежи. Они разделяют общие операции над последовательностями, такие как индексация, конкатенация и нарезание, но поддерживают также специфические для типа вызовы методов. Связанным термином “итерируемый” обозначается либо физическая последовательность, либо виртуальная последовательность, которая производит свои элементы по требованию.
5. Термином “отображение” обозначается объект, который отображает ключи на связанные значения. Словарь Python – единственный тип отображения в наборе основных типов. Отображения не поддерживают позиционное упорядочение слева направо; они поддерживают доступ к хранящимся данным по ключу плюс специфические для типа вызовы методов.
6. “Полиморфизм” означает, что смысл операции (вроде +) зависит от объектов, на которых она выполняется. Выясняется, что это (*вероятно*) ключевая идея правильного использования Python – отказ от ограничения кода конкретными типами делает этот код применимым ко многим типам.

Числовые типы

В этой главе начинается наш углубленный тур по языку Python. Данные в Python принимают форму *объектов* — либо встроенных объектов, предоставляемых Python, либо объектов, которые мы создаем с применением инструментов Python и других языков, таких как C. На самом деле объекты являются фундаментом каждой программы Python, какую только придется писать. Поскольку объекты представляют собой настолько фундаментальное понятие в программировании на Python, мы уделяем им внимание первыми.

В предыдущей главе мы кратко прошлись по основным типам объектов Python. Несмотря на то что в ней были введены важные термины, мы избегали раскрытия слишком многих особенностей в интересах экономии пространства. Здесь мы приступим к более тщательной вторичной ревизии концепций типов данных, чтобы восполнить недостающие детали. Давайте начнем с исследования первой категории типов данных: числовых типов Python и операций над ними.

Основы числовых типов

Большинство числовых типов Python довольно обычны и вероятно покажутся знакомыми, если в прошлом вы использовали почти любой другой язык программирования. Они могут применяться для отслеживания банковского сальдо, расстояния до Марса, количества посетителей веб-сайта и практически любой другой числовой величины.

На самом деле числа в Python являются не одиночным типом объектов, а категорией похожих типов. Python поддерживает привычные числовые типы (целые числа и числа с плавающей точкой), литералы для создания чисел и выражения для их обработки. Вдобавок Python предоставляет расширенную поддержку численного программирования и объекты для более сложной работы. Полный перечень численного инструментального набора Python включает:

- объекты целых чисел и чисел с плавающей точкой;
- объекты комплексных чисел;
- десятичные числа: объекты с фиксированной точностью;
- дроби: объекты рациональных чисел;
- множества: коллекции с числовыми операциями;
- булевские значения: “истина” и “ложь”;
- встроенные функции и модули: `round`, `math`, `random` и т.д.;

- выражения; неограниченная точность целых чисел; побитовые операции; шестнадцатеричный, восьмеричный и двоичный форматы;
- сторонние расширения: векторы, библиотеки, визуализация, вычерчивание и т.д.

Из-за того, что типы в первом пункте списка, как правило, встречаются в большинстве действий кода Python, мы сначала рассмотрим базовые числа и основы, после чего перейдем к исследованию других типов из списка, исполняющих специализированные роли. Мы здесь изучим также *множества*, которые обладают качествами и чисел, и коллекций, но в целом считаются больше первыми, нежели вторыми. Однако прежде чем заняться кодом, в последующих нескольких разделах будет представлен обзор того, как мы записываем и обрабатываем числа в сценариях.

Числовые литералы

Среди своих базовых типов Python предоставляет *целые числа*, которые могут быть положительными и отрицательными, и числа с *плавающей точкой*, имеющие дробную часть. Кроме того, Python позволяет записывать целые числа с использованием шестнадцатеричных, восьмеричных и двоичных литералов, предлагает тип комплексных чисел и разрешает целым числам иметь неограниченную *точность* – они могут разрастаться до такого количества цифр, которое только способно уместиться в доступном пространстве памяти. В табл. 5.1 показано, как выглядят числовые типы Python, когда записываются в программах в виде литералов или вызовов функций конструкторов.

Таблица 5.1. Числовые литералы и конструкторы

Литерал	Толкование
1234, -24, 0, 9999999999999999	Целые числа (неограниченный размер)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Числа с плавающей точкой
0o177, 0x9ff, 0b101010	Восьмеричный, шестнадцатеричный и двоичный литералы в Python 3.X
0177, 0o177, 0x9ff, 0b101010	Восьмеричный, восьмеричный, шестнадцатеричный и двоичный литералы в Python 2.X
3+4j, 3.0+4.0j, 3J	Литералы комплексных чисел
set('spam'), {1, 2, 3, 4}	Множества: формы создания Python 2.X и 3.X
Decimal('1.0'), Fraction(1, 3)	Типы расширений десятичных и дробных чисел
bool(X), True, False	Булевский тип и константы

В общем, литералы числовых типов Python прямолинейны в написании, но здесь полезно подчеркнуть несколько концепций кодирования.

Литералы целых чисел и чисел с плавающей точкой

Целые числа записываются в виде строк десятичных цифр. Числа с плавающей точкой имеют десятичную точку и/или необязательный показатель степени со знаком, вводимый посредством символа e или E, за которым следует необязательный знак. Если вы запишете число с десятичной точкой или показателем степени, то Python делает его объектом числа с плавающей точкой и применяет математику с плавающей точкой (не целочисленную), когда данный объект участвует в выражении. В стандартном CPython числа с плавающей точкой реа-

лизованы как числа с плавающей точкой двойной точности (double) языка С и по этой причине получают точность, обеспечиваемую для чисел double компилятором С, который использовался для построения интерпретатора Python.

Целые числа в Python 2.X: нормальные и длинные

В Python 2.X существуют два типа целых чисел – нормальные (часто 32 бита) и длинные (неограниченная точность); чтобы число стало длинным целым, оно должно заканчиваться на `l` или `L`. Поскольку целые числа автоматически преобразуются в длинные целые, когда их величины не умещаются в выделенные биты, самостоятельно набирать букву `L` вам не придется – если нужна повышенная точность, то Python выполнит преобразование.

Целые числа в Python 3.X: единственный тип

В Python 3.X нормальные и длинные типы целых чисел были объединены – есть один тип, который автоматически поддерживает неограниченную точность отдельного длинного типа целых чисел Python 2.X. В результате целые числа могут больше не записываться с хвостовой буквой `l` или `L` и они больше не выводятся с данным символом. За исключением этого большинство программ остались незатронутыми таким изменением, если только в них не выполняется проверка типов на предмет длинных целых Python 2.X.

Шестнадцатеричные, восьмеричные и двоичные литералы

Целые числа могут кодироваться в десятичной (с основанием 10), шестнадцатеричной (с основанием 16), восьмеричной (с основанием 8) или двоичной (с основанием 2) системе счисления, последние три из которых распространены в ряде областей программирования. Литералы шестнадцатеричных чисел начинаются с символов `0x` или `0X`, за которыми следует строка шестнадцатеричных цифр (0–9 и A–F). Шестнадцатеричные цифры могут записываться в нижнем или верхнем регистре. Литералы восьмеричных чисел начинаются с символов `0o` или `0O` (ноль и буква `o` в нижнем или верхнем регистре), за которыми следует строка восьмеричных цифр (0–7). В Python 2.X литералы восьмеричных чисел могут также записываться только с ведущим 0, но это не так в Python 3.X – первоначальную восьмеричную форму слишком легко спутать с десятичной формой, а потому она заменена новым форматом `0o`, который можно применять в Python 2.X, начиная с версии 2.6. Литералы двоичных чисел, появившиеся в версиях 2.6 и 3.0, начинаются с символов `0b` или `0B`, за которыми следуют двоичные цифры (0–1).

Обратите внимание, что все описанные литералы производят объекты целых чисел в программном коде; они являются всего лишь альтернативным синтаксисом для указания значений. Вызовы встроенных функций `hex(I)`, `oct(I)` и `bin(I)` преобразуют целое число в строку представления в трех системах счисления, а `int(str, base)` преобразует строку в целое число по заданному основанию `base`.

Комплексные числа

Литералы комплексных чисел Python записываются как `действительная_часть+мнимая_часть`, где `мнимая_часть` заканчивается символом `j` или `J`. Формально `действительная_часть` необязательна, так что `мнимая_часть` может появляться сама по себе. Внутренне комплексные числа реализованы

в виде пары чисел с плавающей точкой, но все числовые операции выполняют математику комплексных чисел, когда применяются к таким числам. Комплексные числа можно также создавать вызовом встроенной функции `complex(real, imag)`.

Кодирование других числовых типов

Как вы увидите позже в главе, существуют дополнительные числовые типы, указанные в конце табл. 5.1, которые исполняют более сложные или специализированные роли. Одни типы будут создаваться вызовом функций из импортированных модулей (например, десятичные и дробные числа), а другие имеют собственный литеральный синтаксис (скажем, множества).

Встроенные инструменты для обработки объектов чисел

Кроме встроенных числовых литералов и вызовов функций конструкторов, приведенных в табл. 5.1, Python предлагает набор инструментов для обработки объектов чисел.

- Операции выражений: `+`, `-`, `*`, `/`, `>>`, `**`, `&` и т.д.
- Встроенные математические функции: `pow`, `abs`, `round`, `int`, `hex`, `bin` и т.д.
- Служебные модули: `random`, `math` и т.д.

Все они встречаются по мере продвижения вперед.

Хотя числа обрабатываются главным образом с помощью выражений, встроенных функций и модулей, в наши дни они располагают несколькими *методами*, специальными для типов, которые также будут встречаться в этой главе. Например, числа с плавающей точкой имеют метод `as_integer_ratio`, полезный для типа дробного числа, и метод `is_integer` для проверки, является ли число целым. Целые числа имеют разнообразные атрибуты, включая введенный в Python 3.1 новый метод `bit_length`, который дает количество битов, необходимых для представления значения объекта. Кроме того, как частично коллекция и частично число, *множества* поддерживают методы и выражения.

Поскольку выражения — наиболее важный инструмент для большинства типов, давайте рассмотрим их следующими.

Операции выражений Python

Возможно, самым фундаментальным инструментом, обрабатывающим числа, является *выражение*: комбинация чисел (или других объектов) и операций, которая вычисляет значение, когда выполняется Python. Выражения в Python записываются с использованием обычных математических обозначений и символов операций. Скажем, для сложения двух чисел `X` и `Y` понадобится записать выражение `X + Y`, которое сообщит Python о необходимости применения операции `+` к значениям, именованным как `X` и `Y`. Результатом выражения будет сумма `X` и `Y`, т.е. еще один числовой объект.

В табл. 5.2 перечислены все операции выражений, доступные в Python. Многие из них не требуют объяснений; например, поддерживаются обычные математические операции (`+`, `-`, `*`, `/` и т.д.). Некоторые операции покажутся знакомыми, если в прошлом вы использовали другие языки: `%` вычисляет остаток от деления, `<<` выполняет побитовый сдвиг влево, `&` вычисляет результат побитового “И” и т.д. Другие операции более специфичны для Python и не все они числовые по своей природе: например,

операция `is` проверяет идентичность объекта (т.е. адрес в памяти, строгая форма равенства), а `lambda` создает неименованную функцию.

Таблица 5.2. Операции выражений Python и старшинство

Операции	Описание
<code>yield x</code>	Протокол <code>send</code> функции генератора
<code>lambda аргументы: выражение</code>	Создание анонимной функции
<code>x if y else z</code>	Тернарный выбор (<code>x</code> оценивается, только если <code>y</code> истинно)
<code>x or y</code>	Логическое “ИЛИ” (<code>y</code> оценивается, только если <code>x</code> ложно)
<code>x and y</code>	Логическое “И” (<code>y</code> оценивается, только если <code>x</code> истинно)
<code>not x</code>	Логическое “НЕ”
<code>x in y, x not in y</code>	Членство (итерируемые объекты, множества)
<code>x is y, x is not y</code>	Проверки идентичности объектов
<code>x < y, x <= y, x > y, x >= y</code>	Сравнение по абсолютной величине, проверка на подмножество и надмножество
<code>x == y, x != y</code>	Операции равенства значений
<code>x y</code>	Побитовое “ИЛИ”, объединение множеств
<code>x ^ y</code>	Побитовое “исключающее ИЛИ”, симметричная разность множеств
<code>x & y</code>	Побитовое “И”, пересечение множеств
<code>x << y, x >> y</code>	Сдвиг <code>x</code> влево или вправо на <code>y</code> битов
<code>x + y</code>	Сложение, конкатенация
<code>x - y</code>	Вычитание, разность множеств
<code>x * y</code>	Умножение, повторение
<code>x % y</code>	Остаток, формат
<code>x / y, x // y</code>	Деление: настоящее и с округлением в меньшую сторону
<code>-x, +x</code>	Противоположность, идентичность
<code>~x</code>	Побитовое “НЕ” (инверсия)
<code>x ** y</code>	Возведение в степень
<code>x[i]</code>	Индексация (последовательности, отображения и т.д.)
<code>x[i:j:k]</code>	Нарезание
<code>x(...)</code>	Вызов (функции, метода, класса и прочих вызываемых объектов)
<code>x.атрибут</code>	Ссылка на атрибут
<code>(...)</code>	Кортеж, выражение, выражение генератора
<code>[...]</code>	Список, списковое включение
<code>{...}</code>	Словарь, множество, включения множеств и словарей

Поскольку в книге рассматриваются Python 2.X и 3.X, ниже приведены примечания, которые касаются отличий между версиями и последних дополнений, связанные с операциями из табл. 5.2.

- В Python 2.X неравенство значений можно записывать либо как `X != Y`, либо как `X <> Y`. В Python 3.X второй вариант был убран по причине избыточности. В любой версии для всех проверок неравенства значений рекомендуется применять `X != Y`.
- В Python 2.X выражение с обратными апострофами `X` работает так же, как `repr(X)` и преобразует объекты в отображаемые строки. По причине своей неясности оно было изъято из Python 3.X; используйте более читабельные встроенные функции `str` и `repr`, описанные в разделе “Форматы числового отображения” далее в главе.
- Выражение деления с округлением в меньшую сторону `X // Y` всегда усекает дробную часть в Python 2.X и 3.X. Выражение `X / Y` выполняет настоящее деление в Python 3.X (предохраняя дробную часть) и классическое деление в Python 2.X (отбрасывая дробную часть для целых чисел). См. раздел “Деление: классическое, с округлением в меньшую сторону и настоящее” далее в главе.
- Синтаксис `[...]` применяется для списковых литералов и выражений списковых включений. В последнем случае выполняется неявный цикл, а результаты выражения накапливаются в новом списке. За примерами обращайтесь в главы 4, 14 и 20.
- Синтаксис `(...)` используется для кортежей и группирования выражений, а также в качестве выражений генераторов – формы спискового включения, которая производит результаты по требованию вместо построения итогового списка. Примеры ищите в главах 4 и 20. Во всех трех контекстах круглые скобки иногда можно опускать. Когда круглые скобки не указаны для кортежа, то запятая, разделяющая его элементы, действует подобно операции с наименьшим старшинством, если не имеет иного значения.
- Синтаксис `{...}` применяется для литералов словарей, а в Python 3.X и 2.7 – для литералов множеств и включений множеств и словарей. Множества раскрываются в этой главе; примеры ищите в главах 4, 8, 14 и 20.
- Операция `yield` и тернарный выбор `if/else` доступны в Python 2.5 и последующих версиях. Операция `yield` возвращает аргументы `send(...)` в генераторах; тернарный выбор `if/else` представляет собой сокращение для многострочного оператора `if`. Операция `yield` требует круглых скобок, если не является единственной в правой части оператора присваивания.
- Операции сравнения можно выстраивать в цепочку: `X < Y < Z` дает такой же результат, как и `X < Y and Y < Z`. См. раздел “Сравнения: нормальные и сцепленные” далее в главе.
- В недавних версиях Python выражение нарезания `X[I:J:K]` представляет собой эквивалент индексации с объектом среза: `X[slice(I, J, K)]`.
- В Python 2.X сравнения по абсолютной величине для разнородных типов разрешены; они преобразуют числа в общий тип и упорядочивают другие разнородные типы в соответствии с именами типов. В Python 3.X сравнения по абсолютной величине для разнородных нечисловых типов запрещены и приводят к генерации исключений, что касается также сортировки одного объекта по содержимому другого объекта.

- В Python 3.X больше не поддерживаются сравнения по абсолютной величине для словарей (хотя проверки на равенство разрешены); одной возможной заменой является `sorted(aDict.items())`.

Вы увидите большинство операций из табл. 5.2 в действии позже, а пока нам необходимо взглянуть на способы объединения этих операций в выражениях.

Смешанные операции следуют старшинству операций

Как и в других языках, сложные выражения в Python записываются путем объединения операций из табл. 5.2. Например, сумму двух произведений можно записать в виде смеси переменных и операций:

`A * B + C * D`

Как Python узнает, какую операцию нужно выполнить первой? Ответ на этот вопрос кроется в *старшинстве операций*. Когда вы записываете выражение, имеющее более одной операции, то Python группирует его части согласно тому, что называется *правилами старшинства*, и такое группирование определяет порядок, в котором вычисляются части выражения. Операции в табл. 5.2 упорядочены по старшинству.

- Операции, находящиеся ниже в табл. 5.2, имеют более высокий приоритет и потому в смешанных выражениях вычисляются раньше.
- Операции, расположенные в одной строке табл. 5.2, обычно группируются слева направо, когда они скомбинированы (за исключением возведения в степень, которое группируется справа налево, и сравнений, выстраиваемых в цепочку слева направо).

Скажем, если вы запишете `X + Y * Z`, то Python первым выполнит умножение (`Y * Z`), а затем сложение результата и `X`, потому что операция `*` имеет более высокий приоритет (находится ниже в табл. 5.2), чем `+`. Аналогично в начальном примере этого раздела оба умножения (`A * B` и `C * D`) будут выполнены раньше сложения их результатов.

Круглые скобки группируют подвыражения

Если вы аккуратно группируете части выражений с помощью круглых скобок, тогда можете полностью забыть о старшинстве. Заключение подвыражений в круглые скобки приводит к переопределению правил старшинства Python; выражения в круглых скобках всегда вычисляются первыми и затем их результаты используются в охватывающих выражениях.

Например, вместо написания `X + Y * Z` вы могли бы записать его в одной из следующих форм, чтобы заставить Python вычислять выражение в желаемом порядке:

`(X + Y) * Z`
`X + (Y * Z)`

В первом случае операция `+` применяется к `X` и `Y` первой, потому что это подвыражение помещено в круглые скобки. Во втором случае первой выполняется операция `*` (в частности, как если бы круглые скобки отсутствовали). Вообще говоря, добавление круглых скобок в крупные выражения – хорошая мысль, т.к. они не только обеспечивают вычисление в желаемом порядке, но и содействуют лучшей читабельности.

Разнородные типы преобразуются

Помимо смешивания операций в выражениях допускается также смешивать числовые типы. Скажем, вы можете сложить целое число и число с плавающей точкой:

```
40 + 3.14
```

Но возникает еще один вопрос: какой тип будет у результата – целый или с плавающей точкой? Ответ прост, особенно если ранее вы использовали почти любой другой язык: в выражениях с разнородными числовыми типами Python сначала преобразует операнды к типу самого сложного операнда и затем выполняет операции над операндами того же самого типа. Такое поведение похоже на преобразования типов в языке C.

Вот как в Python оценивается сложность числовых типов: целые числа проще чисел с плавающей точкой, которые в свою очередь проще комплексных чисел. Таким образом, когда целое число смешивается с числом с плавающей точкой, как в предыдущем примере, то целое число преобразуется в значение с плавающей точкой и математика с плавающей точкой выдает результат с плавающей точкой:

```
>>> 40 + 3.14      # Преобразование целого в число с плавающей точкой,  
                  # математика и результат с плавающей точкой
```

```
43.14
```

Точно так же любое выражение с разнородными типами, где один операнд является комплексным числом, приводит к преобразованию другого операнда в комплексное число и получению результата – комплексного числа. В Python 2.X нормальные целые числа также преобразуются в длинные целые всякий раз, когда их значения оказываются слишком большими, чтобы уместиться в нормальное целое; в Python 3.X все целые числа относятся к длинным целым.

Вызывая встроенные функции, типы можно преобразовывать вручную:

```
>>> int(3.1415)      # Сокращает число с плавающей точкой до целого  
3  
>>> float(3)        # Преобразует целое число в число с плавающей точкой  
3.0
```

Тем не менее, как правило, в этом нет нужды: поскольку Python автоматически преобразовывает операнды в более сложный тип внутри выражения, результаты оказываются обычно теми, которые желательны.

Кроме того, имейте в виду, что все преобразования между разнородными типами применяются только при смешивании в выражении *числовых* типов (например, целого и с плавающей точкой), в том числе с использованием числовых операций и операций сравнения. В общем случае Python не выполняет автоматическое преобразование между другими типами. Скажем, добавление строки к целому числу приводит к ошибке, если только не преобразовать одно в другое вручную; дождитесь примера в главе 7, когда мы займемся строками.



В Python 2.X разнородные нечисловые типы могут сравниваться, но никаких преобразований не происходит – разнородные типы сравниваются в соответствии с правилом, которое выглядит детерминированным, но эстетически непривлекательным: оно предусматривает сравнение строковых имен типов объектов. В Python 3.X сравнения по абсолютной величине для разнородных нечисловых типов запрещены и вызывают исключения. Обратите внимание, что это применяется только к операциям сравнения, таким как `>`; использовать другие операции вроде `+` с разнородными нечисловыми типами не разрешено ни в Python 3.X, ни в Python 2.X.

Обзор: перегрузка операций и полиморфизм

Хотя прямо сейчас наше внимание сосредоточено на встроенных числах, следует отметить, что все операции Python могут быть перегружены (т.е. реализованы) в классах Python и типах расширений C, чтобы работать с объектами, которые вы создаете. Например, позже вы увидите, что с объектами, создаваемыми посредством классов, можно выполнять сложение и конкатенацию с помощью выражений `x+y`, индексировать посредством выражений `x[i]` и т.д.

Более того, сам Python автоматически перегружает некоторые операции, так что они производят разные действия в зависимости от типа обрабатываемых встроенных объектов. Например, операция `+` выполняет сложение, когда применяется к числам, но конкатенацию в случае применения к объектам последовательностей, таким как строки и списки. На самом деле `+` может означать что угодно, когда применяется к объектам, определяемым через классы.

Как упоминалось в предыдущей главе, такое свойство обычно называется *полиморфизмом* – термин, указывающий на то, что смысл операции зависит от типа объектов, на которых она выполняется. Мы снова вернемся к концепции полиморфизма во время исследования функций в главе 16, потому что в таком контексте он станет гораздо яснее.

Числа в действии

Займемся написанием кода! Пожалуй, наилучший способ понять числовые объекты и выражения – посмотреть на них в действии, поэтому имея под рукой изложенные ранее основы, запустим интерактивный сеанс и опробуем ряд простых, но иллюстративных операций (запуск интерактивного сеанса обсуждается в главе 3).

Переменные и базовые выражения

Прежде всего, давайте поупражняемся с базовой математикой. В следующем взаимодействии мы сначала присваиваем двум *переменным* (`a` и `b`) целые числа, так что их можно будет использовать позже в более крупном выражении. Переменные представляют собой просто имена, создаваемые вами или Python, которые применяются для отслеживания информации в программе. В следующей главе мы обсудим их более подробно, но в Python:

- переменные создаются, когда им впервые присваиваются значения;
- переменные заменяются своими значениями, когда используются в выражениях;
- переменным должны быть присвоены значения до того, как их можно будет применять в выражениях;
- переменные ссылаются на объекты и никогда не объявляются заранее.

Другими словами, показанные далее присваивания автоматически вызывают появление переменных `a` и `b`:

```
% python
>>> a = 3      # Имя создается: не объявляется заранее
>>> b = 4
```

Здесь также используется *комментарий*. Вспомните, что в коде Python текст, находящийся после символа `#` и продолжающийся до конца строки, считается комментарием и игнорируется Python. Комментарии – это способ написания документации по коду, предназначенному для чтения человеком, и важная часть программирования. Я добав-

ляю их в большинство примеров, приводимых в книге, чтобы помочь лучше понять код. В следующей части книги мы ознакомимся со связанным, но более функциональным средством – строками документации, которые прикрепляют текст комментариев к объектам, так что он становится доступным после загрузки кода.

Однако поскольку набираемый интерактивно код является временным, в таком контексте обычно нет необходимости записывать комментарии. Если вы в ходе чтения прорабатываете примеры, то это означает, что вам не нужно набирать текст комментария, начиная с символа `#` и вплоть до конца строки; при работе в интерактивном сеансе такие части операторов необязательны.

Теперь задействуем новые объекты целых чисел в ряде выражений. В данный момент `a` и `b` имеют значения 3 и 4 соответственно. Переменные подобного рода заменяются своими значениями всякий раз, когда встречаются внутри выражения, и при интерактивной работе результаты выражений тотчас же выводятся:

```
>>> a + 1, a - 1      # Сложение (3 + 1), вычитание (3 - 1)
(4, 2)
>>> b * 3, b / 2      # Умножение (4 * 3), деление (4 / 2, результат Python 3.X)
(12, 2.0)
>>> a % 2, b ** 2      # Деление по модулю (нахождение остатка),
#   возведение в степень (4 ** 2)
(1, 16)
>>> 2 + 4.0, 2.0 ** b # Преобразования разнородных типов
(6.0, 16.0)
```

Формально выводимыми результатами являются *кортежи* из двух значений, потому что строки, набираемые в приглашении к вводу, содержат два выражения, разделенные запятой; именно потому результаты отображаются в круглых скобках (кортежи более подробно рассматриваются позже). Важно понимать, что выражения работают из-за того, что присутствующим в них переменным `a` и `b` были присвоены значения. Если вы укажете другую переменную, которой *еще не присваивалось* значение, то Python сообщит об ошибке, а не заполнит переменную каким-либо стандартным значением:

```
>>> c * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
Ошибка в имени: имя c не определено
```

Предварительно объявлять переменные в Python не требуется, но им хотя бы раз должны быть присвоены значения, прежде чем их можно будет применять. На практике это означает, что вам придется инициализировать счетчики нулем до того, как их можно будет увеличивать, инициализировать списки пустым списком, прежде чем их можно будет дополнять, и т.д.

Ниже приведены два чуть более крупных выражения, которые иллюстрируют группирование операций и преобразования, а также отличие в операции деления в Python 3.X и 2.X:

```
>>> b / 2 + a          # То же, что и ((4 / 2) + 3) [используйте 2.0 в Python 2.X]
5.0
>>> b / (2.0 + a)      # То же, что и (4 / (2.0 + 3)) [используйте print
#   в версиях, предшествующих Python 2.7]
0.8
```

В первом выражении скобки отсутствуют, поэтому Python автоматически группирует компоненты согласно правилу старшинства – из-за того, что операция `/` находится в табл. 5.2 ниже `+`, ее приоритет выше, и она вычисляется первой. Результат получается такой, как если бы выражение было организовано с помощью скобок в соответствие с тем, что показано в комментарии справа от кода.

Кроме того, обратите внимание на то, что в первом выражении все числа являются *целыми*. В итоге операция `/` из Python 2.X выполняет целочисленное деление и сложение, давая результат 5, тогда как операция `/` из Python 3.X выполняет настоящее деление, которое всегда предохраняет дробные остатки и дает показанный результат 5.0. Если вам необходимо целочисленное деление Python 2.X в версии Python 3.X, тогда запишите выражение в виде `b // 2 + a`. Если же нужно настоящее деление Python 3.X в версии Python 2.X, то запишите выражение в форме `b / 2.0 + a` (мы обсудим деление очень скоро).

Во втором выражении часть `+` помещена в круглые скобки, чтобы заставить Python вычислить ее первой (т.е. перед операцией `/`). Мы также сделали один из операндов числом с плавающей точкой, добавив десятичную точку: `2.0`. По причине разнородности типов Python преобразует целое значение, на которое ссылается `a`, в значение с плавающей точкой (3.0) перед выполнением операции `+`. Если бы все числа в выражении были целыми, то целочисленное деление `(4 / 5)` выдало бы усеченное целое значение 0 в Python 2.X, но значение с плавающей точкой 0.8 в Python 3.X. Дождитесь формальных деталей, касающихся деления.

Форматы числового отображения

Если вы работаете с Python 2.6, Python 3.0 или более ранней версией, то результат последнего из предшествующих примеров на первый взгляд может показаться несколько странным:

```
>>> b / (2.0 + a)          # Python <= 2.6: выводит больше (или меньше) цифр
0.8000000000000004
>>> print(b / (2.0 + a))  # Но print выполняет округление
0.8
```

Мы соприкоснулись с таким явлением в предыдущей главе, и в Python 2.7, 3.1 и последующих версиях оно отсутствует. Причина столь странного результата связана с ограничениями оборудования для вычислений с плавающей точкой и его неспособностью к точному представлению некоторых значений в ограниченном количестве битов. Поскольку архитектура компьютера выходит за рамки тематики настоящей книги, мы обойдем это и заявим, что оборудование для вычислений с плавающей точкой делает все возможное, и ни оно, ни Python здесь не ошибается.

В действительности возникает просто проблема *отображения* – автоматический вывод интерактивной подсказки содержит больше цифр, чем вывод оператора `print`, только потому, что он использует отличающийся алгоритм. В памяти хранится одно и то же число. Если вы не хотите видеть все цифры, тогда применяйте `print`; как объясняется во врезке “Форматы отображения `str` и `repr`” далее в главе, вы получите отображение, дружественное в отношении пользователя. Начиная с версий Python 2.7 и 3.1, логика отображения чисел с плавающей точкой старается быть более интеллектуальной и обычно показывает меньше цифр, но временами больше.

Тем не менее, обратите внимание, что так много цифр отображается не для всех значений:

```
>>> 1 / 2.0
0.5
```

и что есть больше способов отображать числа, нежели использование `print` и автоматического вывода (следующий код выполнялся в Python 3.3, и его вывод может слегка варьироваться в более старых версиях):

```
>>> num = 1 / 3.0
>>> num                                # Автоматический вывод
0.3333333333333333
>>> print(num)                          # Явный вывод
0.3333333333333333
>>> '%e' % num                         # Выражение форматирования строк
'3.333333e-01'
>>> '%.2f' % num                        # Альтернативный формат с плавающей точкой
'0.33'
>>> '{0:.2f}'.format(num)               # Метод форматирования строк: Python 2.6, 3.0
# и последующие версии
'0.33'
```

В последних трех выражениях задействовано *форматирование строк* – инструмент, обеспечивающий гибкость форматирования, который мы исследуем в главе 7, посвященной строкам. Его результатами являются строки, которые обычно пригодны для отображения и отчетов.

Форматы отображения `str` и `repr`

Формально отличие между стандартным интерактивным выводом и выводом `print` обусловлено отличием между встроенными функциями `repr` и `str`:

```
>>> repr('spam')      # Используется интерактивным выводом: форма как в коде
"'spam'"
>>> str('spam')       # Используется print: форма, дружественная к пользователю
'spam'
```

Они обе преобразуют произвольные объекты в их строковые представления: `repr` (и стандартный интерактивный вывод) производит результаты, которые выглядят, как если бы они были кодом; `str` (и `print`) осуществляет преобразование в более дружественный к пользователю формат, когда он доступен. Некоторые объекты располагают обоими средствами – `str` для обычного применения и `repr` для вывода дополнительных деталей. Данное понятие вновь всплывает на поверхность, когда мы займемся изучением строк и перегрузки операций в классах, и в целом вы узнаете больше о встроенных функциях `repr` и `str` позже в книге.

Помимо предоставления отображаемых строк для произвольных объектов встроенная функция `str` также дает имя строковому типу данных. В Python 3.X она может вызываться с именем кодировки для декодирования строки Unicode из байтовой строки (например, `str(b'xy', 'utf8')`) и служит альтернативой методу `bytes.decode`, с которым мы встречались в главе 4. Последнюю расширенную роль мы исследуем в главе 37.

Сравнения: нормальные и сцепленные

До сих пор мы имели дело со стандартными числовыми операциями (сложение и умножение), но числа, как и все объекты Python, можно также сравнивать. Нормальные сравнения работают для чисел вполне ожидаемым образом – они сравнивают относительные величины своих операндов и возвращают булевский результат, который мы обычно проверяем и предпринимаем действие в более крупном операторе и программе:

```
>>> 1 < 2          # Меньше
True
>>> 2.0 >= 1     # Больше или равно: число разнородного типа 1 преобразуется в 1.0
True
>>> 2.0 == 2.0    # Равенство значений
True
>>> 2.0 != 2.0    # Неравенство значений
False
```

Снова обратите внимание на то, что в числовых выражениях (и только в них) разрешены разнородные типы; во втором выражении Python сравнивает значения в рамках более сложного типа с плавающей точкой.

Интересно отметить, что Python также позволяет *выстраивать в цепочку* множество сравнений для выполнения проверок вхождения в диапазон. Сцепленные сравнения являются своего рода краткой записью для более крупных булевых выражений. Словом, Python дает возможность связывать вместе проверки со сравнениями по абсолютной величине, чтобы представлять сцепленные сравнения, такие как проверки вхождения в диапазон. Например, выражение ($A < B < C$) проверяет, находится ли B между A и C ; оно эквивалентно булевой проверке ($A < B$ and $B < C$), но легче воспринимается на глаз (и проще набирается на клавиатуре). Например, пусть есть следующие присваивания:

```
>>> X = 2
>>> Y = 4
>>> Z = 6
```

Приведенные далее два выражения дают идентичные результаты, но первое короче в наборе и может выполняться чуть быстрее, т.к. Python придется оценивать Y только один раз:

```
>>> X < Y < Z    # Сцепленные сравнения: проверки вхождения в диапазон
True
>>> X < Y and Y < Z
True
```

Такая же эквивалентность сохраняется для ложных результатов, к тому же разрешено выстраивать цепочки произвольной длины:

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False
>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

В сцепленных проверках можно использовать другие сравнения, но результирующие выражения могут стать менее понятными, если только вы не будете их оценивать так, как делает Python. Скажем, следующее выражение дает ложный результат, потому что 1 не равно 2:

```
>>> 1 == 2 < 3    # То же, что и 1 == 2 and 2 < 3
False      # Не то же, что и False < 3 (эквивалентно 0 < 3, которое истинно!)
```

Python не сравнивает результат `False` выражения $1 == 2$ с числом 3 — формально это означает то же, что и выражение $0 < 3$, которое дает `True` (как мы увидим позже в главе, `True` и `False` представляют собой всего лишь настроенные 1 и 0).

И последнее замечание, прежде чем двигаться дальше: оставив в стороне выстраивание в цепочку, числовые сравнения основаны на абсолютных величинах и в большинстве случаев просты, хотя числа с плавающей точкой могут не всегда работать ожидаемым образом и для содержательного сравнения требовать преобразований или других манипуляций:

```
>>> 1.1 + 2.2 == 3.3      # Разве это не должно быть True?  
False  
>>> 1.1 + 2.2            # Близко к 3.3, но не точно: ограниченная точность  
3.3000000000000003  
>>> int(1.1 + 2.2) == int(3.3)    # Нормально в случае преобразования:  
# см. также round, floor, trunc  
True                      # Здесь могут также помочь десятичные и дробные числа
```

Причина связана с тем фактом, что числа с плавающей точкой не способны представлять определенные значения точно из-за ограниченного количества битов – фундаментальная проблема в численном программировании, не уникальная для языка Python. Мы обсудим ее позже при рассмотрении *десятичных и дробных* чисел, представляющих собой инструменты, которые могут обойти такие ограничения. Теперь продолжим наш экскурс в основные числовые операции Python, взглянув более детально на деление.

Деление: классическое, с округлением в меньшую сторону и настоящее

В предшествующих разделах вы уже видели, как работает деление, а потому должны знать, что в версиях Python 3.X и 2.X оно ведет себя несколько иначе. На самом деле существуют три разновидности деления и две разных операции деления, одна из которых в Python 3.X изменилась. История заслуживает более пристального внимания, т.к. это является еще одним значительным изменением в Python 3.X и способно нарушить работу кода Python 2.X, а потому давайте разберемся с фактами, касающимися операции деления.

X / Y

Классическое и настоящее деление. В Python 2.X такая операция выполняет *классическое* деление, усекая результат для целых чисел и сохраняя остатки (т.е. дробные части) для чисел с плавающей запятой. В Python 3.X она выполняет *настоящее* деление, всегда сохраняя остатки в результатах с плавающей запятой независимо от типов операндов.

X // Y

Деление с округлением в меньшую сторону. Появившаяся в Python 2.2 и доступная Python 2.X и 3.X, данная операция всегда усекает дробные остатки, округляя в меньшую сторону независимо от типов операндов. Тип результата зависит от типов операндов.

Настоящее деление было добавлено для учета того факта, что результаты первоначальной классической модели деления зависели от типов операндов, и потому их может быть трудно предугадывать в динамически типизируемом языке наподобие Python. Из-за такого ограничения классическое деление было удалено в версии Python 3.X – операции / и // реализуют настоящее деление и деление с округлением в меньшую сторону. По умолчанию в Python 2.X применяются классическое деление и деление с округлением в меньшую сторону, но как вариант доступно настоящее деление.

Подведем итоги сказанному.

- В Python 3.X операция `/` теперь всегда выполняет настоящее деление, возвращая результат с плавающей точкой, который включает любой остаток независимо от типов operandов. Операция `//` выполняет деление с округлением в меньшую сторону, которое усекает остаток и возвращает целое число для целочисленных operandов или число с плавающей точкой, если любой из operandов имеет тип с плавающей точкой.
- В Python 2.X операция `/` делает классическое деление, выполняя целочисленное деление с усечением, если оба operandы являются целыми, и деление с плавающей точкой (с сохранением остатка) в противном случае. Операция `//` делает деление с округлением в меньшую сторону и работает так же, как в версии Python 3.X, выполняя деление с усечением для целых чисел и деление с округлением в меньшую сторону для чисел с плавающей точкой.

Ниже демонстрируется работа двух операций в Python 3.X и 2.X – первая операция в каждом наборе показывает критически важное отличие, которое может повлиять на код:

```
C:\code> C:\Python33\python
>>>
>>> 10 / 4      # В Python 3.X отличается: сохраняет остаток
2.5
>>> 10 / 4.0    # В Python 3.X работает так же: сохраняет остаток
2.5
>>> 10 // 4     # В Python 3.X работает так же: усекает остаток
2
>>> 10 // 4.0   # В Python 3.X работает так же: округляет в меньшую сторону
2.0

C:\code> C:\Python27\python
>>>
>>> 10 / 4      # После переноса в Python 3.X работа этого может нарушиться!
2
>>> 10 / 4.0
2.5
>>> 10 // 4     # В Python 2.X используйте это, если нужно усечение
2
>>> 10 // 4.0
2.0
```

Обратите внимание, что тип данных результата для операции `//` в Python 3.X по-прежнему зависит от типов operandов: если один из них имеет тип с плавающей точкой, тогда результат получит тип с плавающей точкой; в противном случае он будет целочисленным. Хотя это может показаться похожим на зависимое от типов поведение операции `/` в Python 2.X, которое послужило причиной его изменения в Python 3.X, тип возвращаемого значения гораздо менее критичен, чем отличия в самом возвращаемом значении.

Кроме того, поскольку операция `//` была предоставлена отчасти как средство совместимости для программ, которые полагаются на целочисленное деление с усечением (и так происходит чаще, чем может казаться), для целых чисел она обязана возвращать целые числа. Применение операции `//` вместо `/` в Python 2.X, когда требуется целочисленное деление с усечением, помогает сделать код совместимым с Python 3.X.

Поддержка обеих версий Python

Несмотря на то что операция `/` ведет себя по-разному в Python 2.X и 3.X, вы все равно можете поддерживать в своем коде обе версии. Если ваши программы полагаются на целочисленное деление с усечением, тогда используйте операцию `//` в Python 2.X и 3.X, как только что упоминалось. Если ваши программы требуют результатов с плавающей запятой с остатками для целых чисел, то применяйте `float`, чтобы во время выполнения в Python 2.X гарантировать то, что один из операндов операции `/` является числом с плавающей запятой:

```
X = Y // Z      # Всегда усекает, всегда возвращает целочисленный
                  # результат для целых в Python 2.X и 3.X
X = Y / float(Z) # Гарантирует деление с плавающей точкой с остатком
                  # в Python 2.X или Python 3.X
```

В качестве альтернативы с помощью импортирования `__future__` вы можете сделать доступным в Python 2.X поведение операции `/` из версии Python 3.X вместо его принудительного обеспечения через преобразования `float`:

```
C:\code> C:\Python27\python
>>> from __future__ import division # Делает доступным поведение / из Python 3.X
>>> 10 / 4
2.5
>>> 10 // 4                      # Целочисленное деление // одинаково в обеих версиях
2
```

Показанный выше специальный оператор `from` сохраняет актуальность до конца сеанса, когда вводится интерактивно, как здесь, и он должен быть первой исполняемой строкой при использовании в файле сценария (увы, в Python мы можем импортировать из будущего, но не из прошлого).

Округление в меньшую сторону или усечение

Следует отметить одну тонкость: операция `//` неофициально называется *усекающим делением*, но более точно говорить о ней как о делении с *округлением в меньшую сторону* – она усекает результат до ближайшего целого значения, которое меньше подлинного результата. Совокупным эффектом оказывается округление с понижением, а не строгое усечение, и это имеет значение при работе с отрицательными числами. Вы можете прояснить для себя указанное отличие с помощью Python-модуля `math` (модули должны импортироваться, прежде чем можно будет задействовать их содержимое; мы обсудим импортирование позже):

```
>>> import math
>>> math.floor(2.5)      # Ближайшее меньшее значение
2
>>> math.floor(-2.5)
-3
>>> math.trunc(2.5)      # Усечение дробной части (в сторону нуля)
2
>>> math.trunc(-2.5)
-2
```

При выполнении операций деления усечение в действительности происходит только для положительных результатов, т.к. оно совпадает с округлением в меньшую сторону; для отрицательных значений им будет округленный результат (на самом деле в обоих случаях производится округление в меньшую сторону, но для положительных чисел округление – то же самое, что и усечение). Вот код для Python 3.X:

```
C:\code> c:\python33\python
>>> 5 / 2, 5 / -2
(2.5, -2.5)

>>> 5 // 2, 5 // -2          # Усекает в меньшую сторону: округляет
                             # до первого меньшего целого
(2, -3)                      # 2.5 становится 2, -2.5 становится -3

>>> 5 / 2.0, 5 / -2.0       # Повторяется для чисел с плавающей точкой,
                             # хотя результат имеет тот же тип
(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0    # Повторяется для чисел с плавающей точкой,
                             # хотя результат имеет тот же тип
(2.0, -3.0)
```

В версии Python 2.X ситуация похожая, но результаты операции / снова отличаются:

```
C:\code> c:\python27\python
>>> 5 / 2, 5 / -2          # Отличается от Python 3.X
(2, -3)

>>> 5 // 2, 5 // -2        # Это и остальное совпадают в Python 2.X и 3.X
(2, -3)

>>> 5 / 2.0, 5 / -2.0      # Это и остальное совпадают в Python 2.X и 3.X
(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0    # Это и остальное совпадают в Python 2.X и 3.X
(2.0, -3.0)
```

Если вас интересует усечение в сторону нуля независимо от знака, то вы всегда можете прогнать результат деления с плавающей точкой через функцию `math.trunc` безотносительно к версии Python (также взгляните настроенную функцию `round` со связанный функциональностью истроенную функцию `int`, которая дает тот же эффект, но не требует импортирования):

```
C:\code> c:\python33\python
>>> import math
>>> 5 / -2                  # Сохраняет остаток
-2.5
>>> 5 // -2                 # Округляет результат в меньшую сторону
-3
>>> math.trunc(5 / -2)       # Усекает вместо округления (то же, что int())
-2

C:\code> c:\python27\python
>>> import math
>>> 5 / float(-2)           # Сохраняет остаток в Python 2.X
-2.5
>>> 5 / -2, 5 // -2         # Округляет результат в меньшую сторону в Python 2.X
(-3, -3)
>>> math.trunc(5 / float(-2)) # Усекает в Python 2.X
-2
```

Почему усечение важно?

В заключение отметим, что если вы используете Python 3.X, то вот краткая сводка по операциям деления для справочных целей:

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)  # Настоящее деление в Python 3.X
(2.5, 2.5, -2.5, -2.5)
```

```
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2) # Деление с округлением  
                                         # в меньшую сторону в Python 3.X  
(2, 2.0, -3.0, -3)  
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)   # Одинаково в Python 3.X и 2.X  
(3.0, 3.0, 3, 3.0)
```

Для пользователей Python 2.X деление работает следующим образом (три выделенных полужирным результата в выводе отличаются от Python 3.X):

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)    # Классическое деление
                                         # в Python 2.X (отличается)
(2, 2.5, -2.5, -3)
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2) # Деление с округлением
                                         # в меньшую сторону в Python 2.X (такое же)
(2, 2.0, -3.0, -3)
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)   # Однаково в Python 2.X и 3.X
(3, 3.0, 3, 3.0)
```

Вполне возможно, что поведение операции `/` из Python 3.X, не предусматривающее усечения, способно нарушить работу значительного количества программ Python 2.X. Вероятно, из-за наследия языка С многие программисты полагаются на усечение при делении для целых чисел, поэтому им придется научиться применять в таких контекстах операцию `//`. На сегодняшний день вы должны поступать так при написании всего нового кода Python 2.X и Python 3.X – в первом случае для обеспечения совместимости с Python 3.X, а во втором случае из-за того, что операция `/` не производит усечение в Python 3.X. Дождитесь примера с циклом `while`, определяющим простые числа, в главе 13 и соответствующего упражнения в конце части IV, которые иллюстрируют разновидность кода, подверженного влиянию изменений в поведении `/`. Кроме того, в главе 25 будет дополнительно обсуждаться команда `from`, используемая в настоящем разделе.

Точность целых чисел

Деление может слегка отличаться между выпусками Python, но все равно оно довольно-таки стандартно. Есть кое-что чуть более экзотическое. Как упоминалось ранее, целые числа Python 3.X поддерживают неограниченный размер:

```
>>> 99999999999999999999999999999999999999 + 1      # Python 3.X
1000000000000000000000000000000000000000000000000
```

В Python 2.X для длинных целых имеется отдельный тип, но в него производится автоматическое преобразование любого числа, которое слишком велико, чтобы храниться как нормальное целое. Следовательно, для применения длинных целых не требуется какой-то специальный синтаксис и единственный способ, которым можно указать, что используются длинные целые Python 2.X, предусматривает их написание с хвостовой буквой L:

```
>>> 9999999999999999999999999999999999999999999999999 + 1      # Python 2.X
10000000000000000000000000000000000000000000000L
```

Целые числа с неограниченной точностью – удобный встроенный инструмент. Например, вы можете применять их для подсчета государственного долга США в центах прямо в Python (если вам интересно и компьютер располагает достаточным объемом памяти). Благодаря целым числам с неограниченной точностью мы также получили возможность возводить 2 в настолько большие степени в примерах главы 3. Ниже представлены случаи для Python 3.X и Python 2.X:

```
>>> 2 ** 200  
1606938044258990275541962092341162602522202993782792835301376  
>>> 2 ** 200  
1606938044258990275541962092341162602522202993782792835301376L
```

Поскольку для поддержки расширенной точности Python обязан проделывать дополнительную работу, целочисленная математика, как правило, существенно медленнее, чем обычно, когда числа становятся крупными. Однако если вам необходима точность, тогда тот факт, что она доступна, наверняка перевесит потерю в производительности.

Комплексные числа

Несмотря на менее частое использование по сравнению с типами, рассмотренными до сих пор, комплексные числа являются отдельным основным типом объектов в Python. Они обычно применяются в инженерных и научных приложениях. Если вам известно, что они собой представляют, то вы знаете, почему они полезны; если же нет, тогда прочтите текущий раздел.

Комплексные числа представляются в виде двух чисел с плавающей точкой – действительной и мнимой частей – и записываются с добавлением суффикса `j` или `J` к мнимой части. Мы также можем записывать комплексные числа с ненулевой действительной частью, складывая две части посредством операции `+`. Скажем, комплексное число с действительной частью 2 и мнимой частью -3 записывается как $2 + -3j$. Вот несколько примеров математики комплексных чисел в работе:

```
>>> 1j * 1j  
(-1+0j)  
>>> 2 + 1j * 3  
(2+3j)  
>>> (2 + 1j) * 3  
(6+3j)
```

Комплексные числа также позволяют извлекать свои части как атрибуты, поддерживают все обычные математические выражения и могут обрабатываться с использованием инструментов из стандартного модуля `cmath` (версия стандартного модуля `math`, предназначенная для работы с комплексными числами). Но поскольку комплексные числа в большинстве областей программирования применяются редко, остальные детали здесь не рассматриваются. За дополнительными сведениями обращайтесь в справочное руководство по языку Python.

Шестнадцатеричная, восьмеричная и двоичная формы записи: литералы и преобразования

В дополнение к обычному десятичному виду, который использовался до сих пор, целые числа Python могут записываться в шестнадцатеричной, восьмеричной и двоичной формах. На первый взгляд эти три формы могут выглядеть чуждыми, но некоторые программисты находят их удобными альтернативами для указания значений, особенно когда важную роль играет отображение значений на байты. Правила записи были кратко представлены в начале главы, а ниже мы взглянем на ряд примеров.

Имейте в виду, что такие литералы являются просто альтернативным синтаксисом для указания значений объекта целого числа. Скажем, следующие литералы, записанные в Python 3.X или Python 2.X, производят нормальные целые числа со значениями, заданными во всех трех системах счисления. Независимо от того, какая система счис-

ления применялась для указания величины целого числа, в памяти оно будет тем же самым:

```
>>> 0o1, 0o20, 0o377 # Восьмеричные литералы: основание 8, цифры 0-7 (3.X, 2.6+)
(1, 16, 255)
>>> 0x01, 0x10, 0xFF      # Шестнадцатеричные литералы: основание 16,
                           # цифры 0-9/A-F (3.X, 2.X)
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111 # Двоичные литералы: основание 2,
                           # цифры 0-1 (3.X, 2.6+)
(1, 16, 255)
```

Здесь восьмеричное значение 0o377, шестнадцатеричное значение 0xFF и двоичное значение 0b11111111 представляют собой десятичное число 255. Каждая цифра F в шестнадцатеричном значении, например, означает 15 в десятичной форме и четырехбитное 1111 в двоичной форме и отражает степень 16. Таким образом, шестнадцатеричное значение 0xFF и другие преобразуются в десятичные значения, как показано далее:

```
>>> 0xFF, (15 * (16 ** 1)) + (15 * (16 ** 0)) # Как шестнадцатеричное
                                                 # число отображается на десятичное
(255, 255)
>>> 0x2F, (2 * (16 ** 1)) + (15 * (16 ** 0))
(47, 47)
>>> 0xF, 0b1111, (1*(2**3) + 1*(2**2) + 1*(2**1) + 1*(2**0))
(15, 15, 15)
```

По умолчанию Python выводит целые значения в десятичной форме (по основанию 10), но предоставляет встроенные функции, которые дают возможность преобразовывать целые числа в строки цифр из других систем счисления, в форме литералов Python; они удобны, когда программы или пользователи ожидают видеть значения в заданной системе счисления:

```
>>> oct(64), hex(64), bin(64)    # Числа => строки цифр
('0o100', '0x40', '0b1000000')
```

Функция `oct` преобразует десятичное число в восьмеричное, `hex` – в шестнадцатеричное и `bin` – в двоичное. Еще одна встроенная функция, `int`, преобразует строку цифр в целое число и принимает необязательный второй аргумент, позволяющий указывать основание системы счисления; она удобна для работы с числами, которые читаются из файлов в виде строк, а не записываются в сценариях:

```
>>> 64, 0o100, 0x40, 0b1000000          # Цифры => числа в сценариях и строки
(64, 64, 64, 64)
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)
>>> int('0x40', 16), int('0b1000000', 2) # Литеральные формы также поддерживаются
(64, 64)
```

Функция `eval`, с которой вы столкнетесь позже в книге, трактует строки, как если бы они были кодом Python. Следовательно, она обеспечивает похожий результат, но обычно выполняется медленнее. Фактически `eval` компилирует и запускает строку как порцию программы, при этом предполагая, что выполняемая строка была получена из *надежного источника* – ловкий пользователь способен отправить строку, которая удалит файлы на компьютере, так что будьте аккуратны с таким вызовом:

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')
(64, 64, 64, 64)
```

Наконец, вы также можете преобразовывать целые числа в специфичные для системы счисления строки с помощью вызовов методов и выражений *форматирования строк*, которые возвращают лишь цифры, а не лiteralные строки Python:

```
>>> '{0:o}, {1:x}, {2:b}'.format(64, 64, 64) # Числа => цифры, Python 2.6+
'100, 40, 1000000'
>>> '%o, %x, %x, %X' % (64, 64, 255, 255) # Аналогично, во всех версиях Python
'100, 40, ff, FF'
```

Форматирование строк более подробно раскрывается в главе 7.

Прежде чем двигаться дальше, нужно сделать два замечания. Во-первых, из начала этой главы пользователи Python 2.X должны помнить, что восьмеричные числа могут записываться просто с *ведущим нулем* (исходный восьмеричный формат в Python):

```
>>> 0o1, 0o20, 0o377      # Новый восьмеричный формат в Python 2.6+
# (такой же, как в Python 3.X)
(1, 16, 255)
>>> 01, 020, 0377        # Старые восьмеричные литералы в Python 2.X
# (ошибка в Python 3.X)
(1, 16, 255)
```

В Python 3.X синтаксис, использованный во втором примере, приводит к возникновению ошибки. Хотя это не ошибка в Python 2.X, осторегайтесь начинать строку цифр с ведущего нуля, если только не намерены действительно записывать восьмеричное значение. Версия Python 2.X будет трактовать ее как представленную в системе счисления с основанием 8, что может не работать ожидаемым вами образом – 010 в Python 2.X всегда равно десятичному значению 8, но не 10 (вопреки тому, что вы могли или не могли подумать!). Именно для достижения гармонии с шестнадцатеричной и двоичной формами восьмеричный формат в Python 3.X был изменен – вы обязаны применять `0o10` в Python 3.X и вероятно должны поступать так в версиях Python 2.6 и 2.7 ради ясности и прямой совместимости с Python 3.X.

Во-вторых, обратите внимание на то, что такие литералы способны производить произвольно длинные целые числа. Например, показанный ниже код создает целое число в шестнадцатеричной форме записи и затем отображает его сначала в десятичном виде и далее в восьмеричном и двоичном видах с помощью функций преобразования (код выполнялся в Python 3.X: в Python 2.X десятичное и восьмеричное числа имели бы завершающую букву *L* для обозначения отдельного длинного типа, а восьмеричное число отображалось бы без буквы *o*):

Продолжая тему двоичных цифр, в следующем разделе обсуждаются инструменты для обработки индивидуальных битов.

Побитовые операции

Помимо нормальных числовых операций (сложение, вычитание и т.п.) в Python поддерживается большинство числовых выражений, доступных в языке С. Сюда

входят операции, которые трактуют целые числа как строки *двоичных битов* и могут пригодиться, если ваш код Python должен иметь дело с такими вещами, как сетевые пакеты, последовательные порты или упакованные двоичные данные, производимые программой C.

Мы не можем здесь подробно останавливаться на основах булевской математики (тем, кто обязан ее использовать, скорее всего, уже известно, как она работает, а другие могут вообще отложить изучение данной темы), но они прямолинейны. Например, ниже демонстрируются выражения Python, выполняющие побитовый сдвиг и булевые операции на целых числах:

```
>>> x = 1          # Десятичное значение 1 в битах выглядит как 0001
>>> x << 2        # Сдвиг влево на 2 бита: 0100
4
>>> x | 2          # Побитовое ИЛИ (один из битов = 1): 0011
3
>>> x & 1          # Побитовое И (оба бита = 1): 0001
1
```

В первом выражении двоичная 1 (по основанию 2, 0001) сдвигается влево на две позиции, чтобы породить двоичную 4 (0100). Последние две операции выполняют двоичное “ИЛИ” для объединения битов ($0001 | 0010 = 0011$) и двоичное “И” для выбора общих битов ($0001 \& 0001 = 0001$). Такие операции побитового маскирования позволяют декодировать и извлекать множество флагов и других значений внутри одиночного целого числа.

Это одна из областей, где поддержка двоичных и шестнадцатеричных чисел в Python, начиная с версий 3.0 и 2.6, становится особенно полезной – она позволяет записывать и инспектировать числа по битовым строкам:

```
>>> x = 0b0001      # Двоичные литералы
>>> x << 2        # Сдвиг влево
4
>>> bin(x << 2)    # Стока двоичных цифр
'0b100'
>>> bin(x | 0b010)  # Побитовое ИЛИ: один из битов = 1
'0b11'
>>> bin(x & 0b1)    # Побитовое И: оба бита = 1
'0b1'
```

Сказанное также справедливо для значений, которые начинают свое существование как шестнадцатеричные литералы или подвергаются базовым преобразованиям:

```
>>> X = 0xFF        # Шестнадцатеричные литералы
>>> bin(X)
'0b11111111'
>>> X ^ 0b10101010  # Побитовое исключающее ИЛИ: один из битов = 1, но не оба
85
>>> bin(X ^ 0b10101010)
'0b1010101'
>>> int('01010101', 2) # Цифры => число: строка в целое по указанному основанию
85
>>> hex(85)         # Число => цифры: строка шестнадцатеричных цифр
'0x55'
```

Также в этом плане версии Python 3.1 и 2.7 предложили для целых чисел новый метод `bit_length`, который позволяет запрашивать количество битов, требующееся для

представления величины числа в двоичном виде. Того же эффекта часто можно достичь, вычитая 2 из длины строки `bin`, которая получена с применением встроенной функции `len`, описанной в главе 4 (для учета ведущих символов `0b`), хотя поступать подобным образом менее эффективно:

```
>>> X = 99
>>> bin(X), X.bit_length(), len(bin(X)) - 2
('0b1100011', 7, 7)
>>> bin(256), (256).bit_length(), len(bin(256)) - 2
('0b100000000', 9, 9)
```

Мы не собираемся здесь слишком глубоко погружаться в “жонглирование битами”. Несмотря на то что побитовые операции поддерживаются, в высокоуровневом языке вроде Python они зачастую не настолько важны, как в низкоуровневых языках, подобных С. Запомните в качестве эмпирического правила: если вы обнаруживаете, что хотите манипулировать битами в Python, то должны подумать о том, тот ли язык вы используете для реализации программы. Как вы увидите в последующих главах, списки, словари и прочие типы данных в Python предоставляют более богатые – и обычно лучшие – способы кодирования информации, чем битовые строки, особенно когда аудитория потребителей ваших данных включает людей.

Другие встроенные инструменты для обработки чисел

В добавок к основным типам объектов Python также предлагает встроенные *функции* и стандартные библиотечные *модули* для обработки чисел. Скажем, встроенные функции `pow` и `abs` вычисляют соответственно степень и абсолютное значение. Ниже приведены примеры применения модуля `math` (который содержит большинство инструментов из математической библиотеки языка С) и нескольких встроенных функций в Python 3.3. Как было описано ранее, в версиях, предшествующих Python 2.7 и 3.1, ряд чисел с плавающей точкой могут отображаться с большим или меньшим количеством цифр:

```
>>> import math
>>> math.pi, math.e
# Общие константы
(3.141592653589793, 2.718281828459045)
>>> math.sin(2 * math.pi / 180)      # Синус, тангенс, косинус
0.03489949670250097
>>> math.sqrt(144), math.sqrt(2)    # Квадратный корень
(12.0, 1.4142135623730951)
>>> pow(2, 4), 2 ** 4, 2.0 ** 4.0 # Возведение в степень
(16, 16, 16.0)
>>> abs(-42.0), sum((1, 2, 3, 4)) # Абсолютное значение, суммирование
(42.0, 10)
>>> min(3, 1, 2, 4), max(3, 1, 2, 4) # Минимум, максимум
(1, 4)
```

Показанная здесь функция `sum` работает на последовательности чисел, а функции `min` и `max` принимают либо последовательность, либо индивидуальные аргументы. Существует множество способов отбрасывания десятичных цифр из чисел с плавающей точкой. Ранее мы сталкивались с усечением и округлением в меньшую сторону; можно также округлять численно и в целях отображения:

```

>>> math.floor(2.567), math.floor(-2.567)      # Округление в меньшую сторону
                                                # (до меньшего целого)
(2, -3)
>>> math.trunc(2.567), math.trunc(-2.567)     # Усечение (отбрасывание
                                                # десятичных цифр)
(2, -2)
>>> int(2.567), int(-2.567)                  # Усечение (преобразование
                                                # в целое число)
(2, -2)
>>> round(2.567), round(2.467), round(2.567, 2) # Округление (версия Python 3.X)
(3, 2, 2.57)
>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567)    # Округление для отображения
                                                # (глава 7)
('2.6', '2.57')

```

Как уже известно, последний оператор производит строки, которые мы обычно выводим, и поддерживает разнообразные варианты форматирования. Как было описано ранее, вторая с конца проверка также выведет `(3, 2, 2.57)` в версиях, предшествующих Python 2.7 и 3.1, если поместить ее в вызов `print` для запрашивания более дружественного к пользователю отображения. Тем не менее, форматирование строк по-прежнему слегка отличается даже в Python 3.X; функция `round` округляет и отбрасывает десятичные цифры, но производит число с плавающей точкой в памяти, тогда как форматирование строк выдает строку, а не число:

```

>>> (1 / 3.0), round(1 / 3.0, 2), ('%.2f' % (1 / 3.0))
(0.3333333333333333, 0.33, '0.33')

```

Интересно отметить, что в Python доступны три способа вычисления *квадратных корней*: с использованием функции из модуля, выражения и встроенной функции (мы обратимся к ним в упражнении и его решении в конце части IV, чтобы посмотреть, какой способ обеспечивает более быстрое выполнение):

```

>>> import math
>>> math.sqrt(144)                      # Функция из модуля
12.0
>>> 144 ** .5                         # Выражение
12.0
>>> pow(144, .5)                      # Встроенная функция
12.0
>>> math.sqrt(1234567890)             # Более крупные числа
35136.41828644462
>>> 1234567890 ** .5
35136.41828644462
>>> pow(1234567890, .5)
35136.41828644462

```

Обратите внимание, что стандартные библиотечные модули вроде `math` требуется импортировать, но встроенные функции, такие как `abs` и `round`, доступны всегда безо всякого импорта. Другими словами, модули являются внешними компонентами, а встроенные функции находятся в подразумеваемом пространстве имен, в котором Python автоматически осуществляет поиск имен, применяемых в программе. Такое пространство имен просто соответствует стандартному библиотечному модулю, называемому `builtins` в Python 3.X (и `__builtin__` в Python 2.X). В частях книги, посвященных функциям и модулям, распознавание имен будет обсуждаться более подробно, но пока, когда вы слышите о модуле, сразу думайте об импортировании.

Стандартный библиотечный модуль `random` также должен импортироваться. Он предлагает инструменты для решения таких задач, как выбор случайного числа с плавающей точкой между 0 и 1 и выбор случайного целого между двумя числами:

```
>>> import random
>>> random.random()
0.5566014960423105
>>> random.random() # Случайные числа с плавающей точкой, целые, выбор, тасование
0.051308506597373515
>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
9
```

В добавок модуль `random` позволяет случайным образом *выбирать* элемент из последовательности и *тасовать* список элементов:

```
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Holy Grail'
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Life of Brian'
>>> suits = ['hearts', 'clubs', 'diamonds', 'spades']
>>> random.shuffle(suits)
>>> suits
['spades', 'hearts', 'diamonds', 'clubs']
>>> random.shuffle(suits)
>>> suits
['clubs', 'diamonds', 'hearts', 'spades']
```

Несмотря на необходимость написания дополнительного кода, чтобы сделать это более реальным, модуль `random` может быть полезен при тасовании карт в играх, случайном выборе изображений в графическом пользовательском интерфейсе для демонстрации слайдов, выполнении статистического моделирования и многом другом. Позже в книге мы задействуем его снова (скажем, в учебном примере перестановок в главе 20), а более подробные сведения ищите в руководстве по стандартной библиотеке Python.

Другие числовые типы

До сих пор в главе мы использовали основные числовые типы Python – целые, с плавающей точкой и комплексные. Их будет достаточно для большинства задач перемалывания чисел, с которыми придется сталкиваться на практике. Однако в состав Python входит и несколько более экзотических числовых типов, заслуживающих здесь краткого рассмотрения.

Десятичные типы

В версии Python 2.4 появился новый основной числовой тип: десятичный объект, формально известный как `Decimal`. С точки зрения синтаксиса десятичные числа создаются путем вызова функции из импортированного модуля, а не за счет выполнения литерального выражения. С функциональной точки зрения десятичные числа подобны числам с плавающей точкой, но имеют фиксированное количество разрядов после десятичной точки. Следовательно, десятичные числа являются значениями с плавающей точкой *фиксированной точности*.

Например, благодаря десятичным числам мы можем иметь значение с плавающей точкой, которое всегда удерживает всего лишь два десятичных разряда после точки. Кроме того, мы можем указывать, каким образом округлять или усекать лишние десятичные разряды за пределами границ объекта. Хотя десятичный тип в общем случае приводит к снижению производительности, если сравнивать его с нормальным типом с плавающей точкой, он хорошо подходит для представления величин с фиксированной точностью вроде денежных сумм и может помочь в достижении лучшей числовой точности.

Основы десятичных чисел

Последнее утверждение заслуживает пояснений. Как было кратко показано во время исследования сравнений, математика с плавающей точкой не совсем точна из-за ограниченного пространства, применяемого для хранения значений. Скажем, следующее выражение должно давать ноль, но это не так. Результат близок к нулю, но здесь недостаточно битов, чтобы обеспечить точность:

```
>>> 0.1 + 0.1 + 0.1 - 0.3      # Python 3.3
5.551115123125783e-17
```

Вывод результата в формате отображения, дружественного к пользователю, не особенно помогает, поскольку аппаратные средства, имеющие отношение к математике с плавающей точкой, по своему существу ограничены в терминах погрешности (также известной как *точность*). Приведенный далее оператор в Python 3.3 дает такой же результат, как и пример выше:

```
>>> print(0.1 + 0.1 + 0.1 - 0.3)      # Ранние версии Python
                                         # (в Python 3.3 вывод отличается)
5.5511151231313e-17
```

Тем не менее, в случае использования десятичных типов результат может быть нулемым:

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

Как здесь видно, мы можем создавать десятичные объекты, вызывая функцию конструктора `Decimal` из модуля `decimal` и передавая ей строки, которые имеют желаемое количество десятичных цифр для результирующего объекта (при необходимости применяя функцию `str` для преобразования значений с плавающей точкой в строки). Когда десятичные объекты с разной точностью смешиваются в выражениях, Python автоматически выполняет преобразование к наибольшему количеству десятичных цифр после точки:

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')
Decimal('0.00')
```

В Python 2.7, 3.1 и последующих версиях также есть возможность создавать десятичный объект из объекта с плавающей точкой посредством вызова вида `decimal.Decimal.from_float(1.25)`, а в последних версиях Python разрешено использовать числа с плавающей точкой напрямую. Преобразование является точным, но иногда выдает большое количество цифр по умолчанию, если только не зафиксировать его, как объясняется в следующем разделе:

```
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)
Decimal('2.775557561565156540423631668E-17')
```

В Python 3.3 и последующих версиях модуль `decimal` был оптимизирован с целью радикального улучшения его производительности: сообщаемый выигрыш в скорости для новой версии составляет от 10 до 100 раз в зависимости от типа тестируемой программы.

Глобальная установка точности для десятичных чисел

С помощью других инструментов из модуля `decimal` можно устанавливать точность для всех десятичных чисел, организовывать обработку ошибок и делать многое другое. Например, объект контекста в этом модуле позволяет указывать точность (количество десятичных цифр) и режимы округления (вниз, вверх и т.д.). Точность применяется глобально ко всем десятичным объектам, создаваемым в вызывающем потоке:

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)          # По умолчанию: 28 цифр
Decimal('0.1428571428571428571428571429')
>>> decimal.getcontext().prec = 4                  # Фиксированная точность
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)    # Ближе к 0
Decimal('1.110E-17')
```

Формально значимость определяется за счет ввода цифр, а точность применяется при выполнении математических операций. Хотя это свойство более тонкое, чем мы можем выяснить в кратком обзоре, оно делает десятичные числа полезными в качестве основы для ряда финансовых приложений и временами может служить альтернативой ручному округлению и форматированию строк:

```
>>> 1999 + 1.33    # Содержит больше цифр в памяти, чем отображается в 3.3
2000.33
>>>
>>> decimal.getcontext().prec = 2
>>> pay = decimal.Decimal(str(1999 + 1.33))
>>> pay
Decimal('2000.33')
```

Диспетчер контекста для десятичных чисел

В Python 2.6, 3.0 и последующих версиях допускается также временно сбрасывать точность с использованием оператора диспетчера контекста `with`. Когда происходит выход из оператора `with`, точность сбрасывается в свое первоначальное состояние. Введите в новом сеансе Python 3.3 (согласно материалам главы 3 здесь ... представляет собой интерактивное приглашение Python для строк продолжения в некоторых интерфейсах и требует ручных отступов; среда IDLE такое приглашение не отображает и делает отступы автоматически):

```
C:\code> C:\Python33\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33333333333333333333333333333333')
>>>
>>> with decimal.localcontext() as ctx:
...     ctx.prec = 2
...     decimal.Decimal('1.00') / decimal.Decimal('3.00')
... 
```

Хотя такой оператор полезен, он требует гораздо большего объема знаний, нежели тот, которым вы располагаете к настоящему моменту; дождитесь описания оператора `with` в главе 34.

Так как на практике десятичный тип применяется относительно редко, я отсылаю вас за дополнительными деталями к руководствам по стандартной библиотеке Python и интерактивной справочной системе. А поскольку десятичный тип решает некоторые из тех же самых проблем с точностью чисел с плавающей точкой, что и дробный тип, давайте посмотрим в следующем разделе, как они соотносятся.

Дробный тип

В версиях Python 2.6 и 3.0 дебютировал новый числового тип `Fraction`, который реализует объект *рационального числа*. По существу он явно хранит числитель и знаменатель, так что избегает ряда неточностей и ограничений математики с плавающей точкой. Подобно десятичным числам дроби не отображаются настолько близко на аппаратные средства компьютера, как числа с плавающей точкой. Это означает, что их производительность может быть не такой хорошей, но также позволяет им приносить дополнительную пользу в стандартном инструменте, когда они обязательны или попросту пригодны.

Основы дробей

Тип `Fraction` – функциональный собрат типа с фиксированной точностью `Decimal`, который был описан в предыдущем разделе, т.к. они оба могут использоваться для решения проблем с числовой неточностью типа с плавающей точкой. Он также применяется похожими способами – как и `Decimal`, тип `Fraction` находится в модуле; чтобы создать объект типа `Fraction`, необходимо импортировать его конструктор и вызвать его, передав числитель и знаменатель (есть и другие способы). Вот пример:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)      # Числитель, знаменатель
>>> y = Fraction(4, 6)      # Упрощается до 2, 3 по наибольшему общему делителю

>>> x
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

После создания объекты Fraction можно использовать в математических выражениях обычным образом:

```
>>> x + y
Fraction(1, 1)
>>> x - y                      # Результаты точны: числитель, знаменатель
Fraction(-1, 3)
>>> x * y
Fraction(2, 9)
```

Объекты Fraction также можно создавать из строк с числами с плавающей точкой, что во многом похоже на десятичные объекты:

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)
>>>
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

Числовая точность дробных и десятичных типов

Обратите внимание, что ситуация отличается от математики с плавающей точкой, которая связана лежащими в основе ограничениями аппаратных средств, имеющих к ней отношение. Для сравнения ниже приведены те же самые операции, выполняемые над объектами с плавающей точкой, и замечания об их ограниченной точности — в последних версиях Python они могут отображать меньше цифр, нежели привычно, но по-прежнему не являются точными значениями в памяти:

```
>>> a = 1 / 3.0          # Результат точен лишь настолько,
                           # насколько позволяют аппаратные средства
>>> b = 4 / 6.0          # Точность может теряться из-за множества вычислений
>>> a
0.3333333333333333
>>> b
0.6666666666666666
>>> a + b
1.0
>>> a - b
-0.3333333333333333
>>> a * b
0.2222222222222222
```

Такая ограниченная точность, присущая математике с плавающей точкой, особенно заметна для значений, которые не могут быть точно представлены по причине ограниченного количества битов в памяти. Типы Fraction и Decimal предлагают способы получения точных результатов, хотя ценой снижения скорости вычислений и более многословного кода. Скажем, в следующем примере (повторенном из предыдущего раздела) числа с плавающей точкой не дают ожидаемый точный нулевой ответ, но типы Fraction и Decimal обеспечивают его:

```
>>> 0.1 + 0.1 + 0.1 - 0.3      # Должен быть нулем (близко, но не точно)
5.551115123125783e-17
>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

Кроме того, дробный и десятичный типы позволяют получать более понятные и точные результаты, чем иногда способен тип с плавающей точкой, разными путями — за счет представления в рациональном виде и за счет ограничения точности:

```

>>> 1 / 3           # Применяйте ".0" в Python 2.X для настоящего деления (/)
0.333333333333333
>>> Fraction(1, 3) # Числовая точность, два пути
Fraction(1, 3)
>>> import decimal
>>> decimal.getcontext().prec = 2
>>> Decimal(1) / Decimal(3)
Decimal('0.33')

```

Фактически дробный тип предохраняет точность и автоматически упрощает результаты. Продолжим предыдущее взаимодействие:

```

>>> (1 / 3) + (6 / 12) # Используйте ".0" в Python 2.X для настоящего деления (/)
0.833333333333333
>>> Fraction(6, 12)   # Результат автоматически упрощается
Fraction(1, 2)
>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)
>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))
Decimal('0.83')
>>> 1000.0 / 1234567890
8.100000073710001e-07
>>> Fraction(1000, 1234567890)    # Значительно проще!
Fraction(100, 123456789)

```

Преобразование дробей и разнородные типы

Для поддержки преобразования дробей объекты с плавающей точкой теперь имеют метод, который выдает их числитель и знаменатель, дробные объекты располагают методом `from_float` и `float` принимает в качестве аргумента объект `Fraction`. В следующем взаимодействии видно, как все это делается (символ * во второй проверке представляет собой специальный синтаксис, который раскрывает кортеж в индивидуальные аргументы; передача аргументов функциям более подробно обсуждается в главе 18):

```

>>> (2.5).as_integer_ratio()                      # Метод объекта с плавающей точкой
(5, 2)
>>> f = 2.5
>>> z = Fraction(*f.as_integer_ratio())          # Преобразует объект с плавающей
                                                #   точкой в дробь: два аргумента
                                                # То же, что и Fraction(5, 2)
>>> z
Fraction(5, 2)
>>> x                                         # x из предыдущего взаимодействия
Fraction(1, 3)
>>> x + z                                     # 5/2 + 1/3 = 15/6 + 2/6
Fraction(17, 6)
>>> float(x)                                    # Преобразует дробь в объект с
                                                #   плавающей точкой 0.333333333333333
>>> float(z)
2.5
>>> float(x + z)
2.833333333333335
>>> 17 / 6
2.833333333333335

```

```
>>> Fraction.from_float(1.75)      # Преобразует объект с плавающей точкой
                                         # в дробь: другой способ
Fraction(7, 4)
>>> Fraction(*(1.75).as_integer_ratio())
Fraction(7, 4)
```

Наконец, в выражениях разрешено смешивать некоторые типы, хотя иногда для предохранения точности требуется преобразовывать в тип Fraction вручную. Взгляните на показанное далее взаимодействие, чтобы понять, как все работает:

```
>>> x
Fraction(1, 3)
>>> x + 2                  # Объект дроби + объект целого -> объект дроби
Fraction(7, 3)
>>> x + 2.0                # Объект дроби + объект с плавающей точкой ->
                                         # объект с плавающей точкой
2.3333333333333335
>>> x + (1./3)             # Объект дроби + объект с плавающей точкой ->
                                         # объект с плавающей точкой
0.6666666666666666
>>> x + (4./3)
1.6666666666666665
>>> x + Fraction(4, 3) # Объект дроби + объект дроби -> объект дроби
Fraction(5, 3)
```

Одно предостережение: несмотря на возможность преобразования числа с плавающей точкой в дробь, в ряде случаев при этом возникает неизбежная потеря точности, потому что число в своей исходной форме с плавающей точкой не является точным. При необходимости такие результаты можно упростить, ограничивая максимальное значение знаменателя:

```
>>> 4.0 / 3
1.3333333333333333
>>> (4.0 / 3).as_integer_ratio() # Точность теряется из-за числа с плавающей точкой
(6004799503160661, 4503599627370496)

>>> x
Fraction(1, 3)
>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())
>>> a
Fraction(22517998136852479, 13510798882111488)
>>> 22517998136852479 / 13510798882111488. # 5 / 3 (или близко!)
1.6666666666666667
>>> a.limit_denominator(10)           # Упростить до ближайшей дроби
Fraction(5, 3)
```

Чтобы узнать больше о типе Fraction, продолжайте экспериментировать самостоятельно и обратитесь к руководствам по стандартным библиотекам Python 2.6, 2.7 и 3.X, а также к другой документации.

Множества

Помимо десятичного типа в Python 2.4 также появился новый тип коллекции, *множество* – неупорядоченная коллекция уникальных и неизменяемых объектов, которая поддерживает операции, соответствующие математической теории множеств. По определению элемент встречается во множестве только однажды независимо от того,

сколько раз он добавлялся. Таким образом, с множествами связаны разнообразные применения, особенно при работе, ориентированной на числа и базы данных.

Поскольку множества являются коллекциями других объектов, они разделяют определенное поведение с такими объектами, как списки и словари, которые выходят за рамки тематики текущей главы. Например, множества итерируемые, могут увеличиваться и уменьшаться по требованию и содержать объекты разнообразных типов. Вы увидите, что множество действует во многом подобно ключам в словаре без значений, но поддерживает добавочные операции.

Однако из-за того, что множества неупорядочены и не отображают ключи на значения, они не являются ни типами последовательностей, ни типами отображений; множества – отдельная категория типов. Кроме того, поскольку множества по существу имеют математическую природу (и многим читателям могут казаться чересчур академическими и использоваться гораздо реже, чем более распространенные объекты вроде словарей), мы исследуем пользу от объектов множеств Python прямо здесь.

Основы множеств в Python 2.6 и предшествующих версиях

В наши дни существует несколько способов создания множеств в зависимости от того, какая версия Python применяется. Учитывая, что книга охватывает все основные версии, давайте начнем со случая для Python 2.6 и предшествующих версий, который также доступен (а временами обязательен) в более поздних версиях Python; вскоре мы уточним это для расширений Python 2.7 и 3.X. Чтобы создать объект множества, вызовите встроенную функцию `set` и передайте ей последовательность или другой итерируемый объект:

```
>>> x = set('abcde')
>>> y = set('bdxyz')
```

Вы получите обратно объект множества, который содержит все элементы из переданного объекта (имейте в виду, что множества неупорядочены позиционно, и потому они не являются последовательностями – их порядок произволен и может варьироваться от выпуска к выпуску Python):

```
>>> x
set(['a', 'c', 'b', 'e', 'd'])    # Формат отображения Python <= 2.6
```

Созданные подобным образом множества поддерживают распространенные математические операции над множествами через операции *выражений*. Обратите внимание, что следующие операции нельзя выполнять на простых последовательностях наподобие строк, списков и кортежей – для применения таких инструментов мы должны создавать множества, передавая их функции `set`:

```
>>> x - y          # Разность
set(['a', 'c', 'e'])
>>> x | y          # Объединение
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])
>>> x & y          # Пересечение
set(['b', 'd'])
>>> x ^ y          # Симметрическая разность (исключающее ИЛИ)
set(['a', 'c', 'e', 'y', 'x', 'z'])
>>> x > y, x < y    # Надмножество, подмножество
(False, False)
```

Заметным исключением из этого правила является проверка членства во множестве `in` – такое выражение определено для работы со всеми остальными типами коллекций, где оно также выполняет проверку членства (или поиск, если вы предпочитаете процедурные термины). Следовательно, для выполнения такой проверки преобразовывать строки и списки во множества не придется:

```
>>> 'e' in x                                # Проверка членства (множества)
True
>>> 'e' in 'Camelot', 22 in [11, 22, 33]    # Но работает также и с другими типами
(True, True)
```

В дополнение к выражениям объект множества предоставляет *методы*, которые соответствуют этим и другим операциям и поддерживают изменения множеств: метод `add` вставляет один элемент, `update` представляет собой объединение на месте, а `remove` удаляет элемент по значению (вызовите `dir` для любого экземпляра множества либо для имени типа `set`, чтобы увидеть все доступные методы). Предположим, что `x` и `y` остались такими, как они были в последнем взаимодействии:

```
>>> z = x.intersection(y)                  # То же, что и x & y
>>> z
set(['b', 'd'])
>>> z.add('SPAM')                         # Вставка одного элемента
>>> z
set(['b', 'd', 'SPAM'])
>>> z.update(set(['X', 'Y']))              # Слияние: объединение на месте
>>> z
set(['Y', 'X', 'b', 'd', 'SPAM'])
>>> z.remove('b')                          # Удаление одного элемента
>>> z
set(['Y', 'X', 'd', 'SPAM'])
```

Как *итеруемые* контейнеры, множества могут также использоваться в операциях вроде `len`, циклах `for` и списковых включениях. Тем не менее, из-за того, что множества неупорядочены, они не поддерживают операции над последовательностями, подобные индексации и нарезанию:

```
>>> for item in set('abc'): print(item * 3)
aaa
ccc
bbb
```

Наконец, хотя показанные ранее выражения с множествами в общем случае требуют двух множеств, их аналоги, основанные на методах, часто способны работать также и с *любым итеруемым типом*:

```
>>> S = set([1, 2, 3])
>>> S | set([3, 4])                      # Выражения требуют двух множеств
set([1, 2, 3, 4])
>>> S | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'
Ошибка типа: неподдерживаемые типы операндов для |: set и list
>>> S.union([3, 4])                      # Но методы разрешают указывать любой итерируемый тип
set([1, 2, 3, 4])
>>> S.intersection([1, 3, 5])
set([1, 3])
>>> S.issubset(range(-5, 5))
True
```

Для получения дополнительных сведений об операциях над множествами обращайтесь в руководство по стандартной библиотеке Python или в какой-нибудь справочник. Хотя операции над множествами можно кодировать вручную с другими типами вроде списков и словарей (и в прошлом так происходило часто), встроенные множества Python применяют эффективные алгоритмы и методики реализации для обеспечения быстрой и стандартной работы.

Литералы множеств в Python 3.X и 2.7

Если вы считаете множества “крутыми”, то со временем они стали заметно “ круче” благодаря новому синтаксису для *литералов* и *включений* множеств, первоначально добавленных только к линейке Python 3.X, но из-за массового спроса перенесенных обратно в версию Python 2.7. В указанных версиях Python для создания объектов множеств мы по-прежнему можем использовать встроенную функцию `set`, но также и новую форму литералов множеств, применяя фигурные скобки, которые раньше были зарезервированы для словарей. В Python 3.X и 2.7 следующие операторы эквивалентны:

```
set([1, 2, 3, 4])      # Вызов встроенной функции (все версии)
{1, 2, 3, 4}           # Более новые литералы множеств (Python 2.7, Python 3.X)
```

Такой синтаксис имеет смысл с учетом того, что множества по существу подобны *словарям без значений* – поскольку элементы множества неупорядочены, уникальны и неизменяемы, они ведут себя очень похоже на ключи словаря. Это функциональное сходство еще более поразительно, учитывая тот факт, что списки ключей словаря в Python 3.X являются объектами *представлений*, которые поддерживают похожие на множества линии поведения, такие как пересечения и объединения (объекты представлений словарей рассматриваются в главе 8).

Независимо от того, как создано множество, версия Python 3.X отображает его с использованием нового литературального формата. В Python 2.7 *принят* новый литературальный синтаксис, но множества *отображаются* все еще с применением формы вывода Python 2.6 из предыдущего раздела. Во всех версиях Python для создания пустых множеств и построения множеств из существующих итерируемых объектов требуется встроенная функция `set`, но для инициализации множеств с известной структурой удобно использовать новую литературальную форму.

Ниже показано, на что похожи множества в Python 3.X. В версии Python 2.7 они такие же за исключением того, что результаты множеств отображаются с помощью системы обозначений `set(...)` из Python 2.X, а порядок следования элементов может варьироваться от версии к версии (во множествах он в любом случае неважен по определению):

```
C:\code> c:\python33\python
>>> set([1, 2, 3, 4])      # Встроенная функция: та же, что и в Python 2.6
{1, 2, 3, 4}
>>> set('spam')          # Добавить все элементы из итерируемого объекта
{'s', 'a', 'p', 'm'}
>>> {1, 2, 3, 4}          # Литералы множеств: новые в Python 3.X (и Python 2.7)
{1, 2, 3, 4}
>>> S = {'s', 'p', 'a', 'm'}
>>> S
{'s', 'a', 'p', 'm'}
>>> S.add('alot')         # Методы работают, как и ранее
>>> S
{'s', 'a', 'p', 'alot', 'm'}
```

Все операции обработки множеств, обсуждавшиеся в предыдущем разделе, в Python 3.X работают аналогично, но результирующие множества выводятся по-другому:

```
>>> S1 = {1, 2, 3, 4}
>>> S1 & {1, 3}          # Пересечение
{1, 3}
>>> {1, 5, 3, 6} | S1    # Объединение
{1, 2, 3, 4, 5, 6}
>>> S1 - {1, 3, 4}      # Разность
{2}
>>> S1 > {1, 3}         # Надмножество
True
```

Обратите внимание, что {} – по-прежнему словарь во всех версиях Python. Пустые множества должны создаваться посредством встроенной функции `set` и выводятся тем же самым способом:

```
>>> S1 - {1, 2, 3, 4}    # Пустые множества выводятся по-другому
set()
>>> type({})
<class 'dict'>
>>> S = set()           # Инициализация пустого множества
>>> S.add(1.23)
>>> S
{1.23}
```

В Python 2.6 и предшествующих версиях множества, созданные с помощью литералов Python 3.X/2.7, поддерживают те же самые методы, часть которых допускает общие итерируемые операнды, не разрешенные в выражениях:

```
>>> {1, 2, 3} | {3, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'
Ошибка типа: неподдерживаемые типы операндов для |: set и list
>>> {1, 2, 3}.union([3, 4])
{1, 2, 3, 4}
>>> {1, 2, 3}.union({3, 4})
{1, 2, 3, 4}
>>> {1, 2, 3}.union(set([3, 4]))
{1, 2, 3, 4}
>>> {1, 2, 3}.intersection((1, 3, 5))
{1, 3}
>>> {1, 2, 3}.issubset(range(-5, 5))
True
```

Ограничение неизменяемости и фиксированные множества

Множества являются мощными и гибкими объектами, но в версиях Python 3.X и 2.X имеют одно ограничение, о котором вы должны помнить – в значительной степени из-за своей реализации множества способны содержать только объекты *неизменяемых* (также известных как “хешируемые”) типов. Следовательно, списки и словари не могут встраиваться во множества, но кортежи могут, если необходимо хранить составные значения. В случае применения в операциях над множествами кортежи сравниваются по их полным значениям:

```

>>> s
{1,23}
>>> S.add([1, 2, 3])      # Во множестве работают только неизменяемые объекты
TypeError: unhashable type: 'list'
Ошибка типа: некхешируемый тип: list
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'
Ошибка типа: некхешируемый тип: dict
>>> S.add((1, 2, 3))
>>> S                      # Списки и словари не допускаются, но кортежи разрешены
{(1, 2, 3), 1, 23}
>>> S | {(4, 5, 6), (1, 2, 3)} # Объединение: то же, что и S.union(...)
{(1, 2, 3), (4, 5, 6), (1, 2, 3)}
>>> (1, 2, 3) in S          # Членство: по полным значениям
True
>>> (1, 4, 3) in S
False

```

Кортежи во множестве могут использоваться, например, для представления дат, записей, IP-адресов и т.п. (кортежи подробно рассматриваются позже в этой части книги). Множества могут также содержать модули, объекты типов и многое другое. Сами множества изменямы и соответственно не могут вкладываться в другие множества напрямую. Если нужно хранить множество внутри другого множества, тостроенная функция `frozenset` работает в точности как `set`, но создает неизменяемое множество, которое не допускает модификацию и потому может быть внедрено в другие множества.

Включения множеств в Python 3.X и 2.7

В дополнение к литералам в версии Python 3.X появилась конструкция включения множеств, которая была также перенесена в версию Python 2.7. Подобно литералам множеств Python 3.X версия Python 2.7 принимает их синтаксис, но отображает результаты с применением системы обозначения множеств Python 2.X. Выражение включения множеств по форме похоже на списковое включение, кратко рассмотренное в главе 4, но записывается в фигурных, а не квадратных скобках и выполняется для создания множества вместо списка. Включения множеств запускают цикл и на каждой итерации накапливают результат выражения; переменная цикла предоставляет доступ к значению текущей итерации для использования в накапливающем выражении. Результатом будет новое множество, созданное за счет выполнения кода, с обычным поведением множества. Ниже демонстрируется включение множества в Python 3.3 (отображение результата и порядок в Python 2.7 будут отличаться):

```

>>> {x ** 2 for x in [1, 2, 3, 4]}    # Включение множества в Python 3.X/2.7
{16, 1, 4, 9}

```

В этом выражении цикл записывается справа, а накапливающее выражение – слева ($x^{**} 2$). Как и для списковых включений, мы получаем по существу то, о чём говорит данное выражение: “предоставить новое множество, содержащее квадраты x для каждого x в списке”. Включения могут также выполнять проход по объектам других видов, таких как строки (первый из приведенных далее примеров иллюстрирует основанный на включениях способ создания множества из существующего итерируемого объекта):

```

>>> {x for x in 'spam'}           # То же, что и set('spam')
{'m', 's', 'p', 'a'}

```

```

>>> {c * 4 for c in 'spam'}      # Множество накопленных результатов выражения
{'pppp', 'aaaa', 'ssss', 'mmmm'}
>>> {c * 4 for c in 'spamham'}
{'pppp', 'aaaa', 'hhhh', 'ssss', 'mmmm'}
>>> S = {c * 4 for c in 'spam'}
>>> S | {'mmmm', 'xxxx'}
{'pppp', 'xxxx', 'mmmm', 'aaaa', 'ssss'}
>>> S & {'mmmm', 'xxxx'}
{'mmmm'}

```

Поскольку остаток включений опирается на концепции, рассматривать которые мы пока еще не готовы, отложим исследование дальнейших деталей до будущих глав книги. В главе 8 мы встретим первого собрата включения множества в Python 3.X и 2.7, включение словаря, а позже обсудим все включения (списков, множеств и словарей), в частности в главах 14 и 20. Вы узнаете, что все включения поддерживают дополнительный синтаксис, не показанный здесь, в том числе вложенные циклы и проверки `if`, которые могут оказаться сложными для понимания до тех пор, пока вы не изучите более крупные операторы.

Для чего используются множества?

С операциями над множествами связан ряд распространенных применений, часть которых имеет больше практический, нежели математический характер. Скажем, из-за того, что элементы сохраняются во множестве только однократно, множества допускается использовать для *фильтрации дубликатов* из коллекций, хотя элементы в ходе процесса могут оказаться переупорядоченными, т.к. в общем случае множества являются неупорядоченными. Коллекцию нужно лишь преобразовать во множество и затем выполнить обратное преобразование (множества разрешено указывать в вызове `list`, потому что они *итерируемые* – еще один технический артефакт, который мы разберем позже):

```

>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L))                                # Удаление дубликатов
>>> L
[1, 2, 3, 4, 5]
>>> list(set(['yy', 'cc', 'aa', 'xx', 'dd', 'aa'])) # Но порядок может измениться
['cc', 'xx', 'yy', 'dd', 'aa']

```

Множества можно также применять для *выделения различий* в списках, строках и других итерируемых объектах (их необходимо просто преобразовать во множества и получить разность), хотя неупорядоченная природа множеств снова означает, что порядок следования результатов может не совпадать с первоначальным порядком. В последних двух примерах ниже сравниваются списки атрибутов в типах строковых объектов для Python 3.X (в случае Python 2.7 результаты будут другими):

```

>>> set([1, 3, 5, 7]) - set([1, 2, 4, 5, 6])      # Найти различия в списках
{3, 7}
>>> set('abcdefg') - set('abdghij')                # Найти различия в строках
{'c', 'e', 'f'}
>>> set('spam') - set(['h', 'a', 'm'])              # Найти различия,
# разнородные типы
{'p', 's'}

```

```
>>> set(dir(bytes)) - set(dir(bytearray))      # В bytes, но не в bytearray
{'__getnewargs__'}
>>> set(dir(bytearray)) - set(dir(bytes))
{'append', 'copy', '__alloc__', '__imul__', 'remove', 'pop', 'insert', ...
и так далее...}
```

Множества также можно использовать для выполнения проверок на *равенство, нейтральное к порядку*, за счет предварительного преобразования во множество, потому что порядок в нем не имеет значения. Говоря более формально, два множества *равны* тогда и только тогда, когда каждый элемент одного множества содержится в другом, т.е. одно является подмножеством другого независимо от порядка. Например, такой прием можно применять для сравнения выводов из программ, которые должны работать одинаково, но способны генерировать результаты, расположенные в разном порядке. Сортировка перед проверкой на равенство обеспечивает аналогичный эффект, но множества не полагаются на затратную сортировку, которая упорядочивает результаты для поддержки дополнительных проверок величин, чего не делают множества (больше, меньше и т.д.):

```
>>> L1, L2 = [1, 3, 5, 2, 4], [2, 5, 3, 4, 1]
>>> L1 == L2                      # В последовательностях порядок имеет значение
False
>>> set(L1) == set(L2)        # Проверка на равенство, нейтральное к порядку
True
>>> sorted(L1) == sorted(L2)    # Похожая проверка, но результаты упорядочены
True
>>> 'spam' == 'asmp', set('spam') == set('asmp'), sorted('spam') == sorted('asmp')
(False, True, True)
```

Множества также можно использовать для отслеживания того, где вы уже были, при обходе графа или другой *циклической* структуры. Скажем, в примерах с транзитивной перезагрузкой модулей и выводом деревьев наследования, которые будут исследоваться в главах 25 и 31 соответственно, посещенные элементы должны отслеживаться во избежание возникновения циклов, как обсуждается в главе 19 с теоретической точки зрения. Применение в данном контексте списка неэффективно, поскольку поиск требует прямолинейного просмотра. Хотя регистрация посещенных состояний в виде ключей в словаре эффективна, множества предлагают альтернативу, которая по существу эквивалентна (и может быть более или менее понятной в зависимости от того, кого вы спрашиваете).

В заключение отметим, что множества удобны и в ситуации, когда приходится иметь дело с крупными наборами данных (например, результатами запросов к базам данных); пересечение двух множеств содержит объекты, общие в обоих множествах, а объединение – все элементы в том и другом множестве. В целях иллюстрации рассмотрим более реалистичный пример работы операций над множествами, применяемых к спискам сотрудников гипотетической компании, с использованием литералов множеств Python 3.X/2.7 и отображения результатов Python 3.X (в Python 2.6 и предшествующих версиях применяйте `set`):

```
>>> engineers = {'bob', 'sue', 'ann', 'vic'}
>>> managers = {'tom', 'sue'}
>>> 'bob' in engineers      # Является ли сотрудник bob инженером (engineer)?
True
>>> engineers & managers # Кто одновременно инженер и менеджер (manager)?
{'sue'}
```

```

>>> engineers | managers          # Все сотрудники в обеих категориях
{'bob', 'tom', 'sue', 'vic', 'ann'}
>>> engineers - managers        # Инженеры, не являющиеся менеджерами
{'vic', 'ann', 'bob'}
>>> managers - engineers       # Менеджеры, не являющиеся инженерами
{'tom'}
>>> engineers > managers        # Все ли менеджеры - инженеры? (надмножество)
False
>>> {'bob', 'sue'} < engineers   # Оба ли сотрудника - инженеры? (подмножество)
True
>>> (managers | engineers) > managers # Все сотрудники - надмножество менеджеров?
True
>>> managers ^ engineers        # Кто находится в одной категории, но не в обеих?
{'tom', 'vic', 'ann', 'bob'}
>>> (managers | engineers) - (managers ^ engineers)    # Пересечение!
{'sue'}

```

Дополнительные детали об операциях над множествами вы можете найти в руководстве по стандартной библиотеке Python и книгах, посвященных математике и базам данных. Кроме того, в главе 8 мы еще вернемся к некоторым рассмотренным здесь операциям над множествами в контексте объектов представлений словарей из Python 3.X.

Булевские значения

Некоторые могут утверждать, что булевский тип в Python, `bool`, является по своей природе числовым, поскольку два его значения, `True` и `False`, являются настроенными версиями целых чисел 1 и 0, которые всего лишь выводят себя по-другому. Хотя это все, что нужно знать большинству программистов, давайте исследуем булевский тип чуть более подробно.

Выражаясь более формально, в наши дни Python имеет явный булевский тип данных по имени `bool` со значениями `True` и `False`, доступными как предварительно определенные имена. Внутренне имена `True` и `False` являются экземплярами типа `bool`, который в свою очередь представляет собой подкласс (в объектно-ориентированном смысле) встроенного целого типа `int`. Имена `True` и `False` ведут себя в точности как целые числа 1 и 0 за исключением того, что они имеют специальную логику вывода — они отображают себя в виде слов `True` и `False`, а не цифр 1 и 0. В типе `bool` это достигается за счет переопределения форматов строк `str` и `repr` для двух указанных объектов.

Вследствие такой настройки вывод булевых выражений, набираемых в интерактивной подсказке, выглядит как `True` и `False` взамен старых и менее очевидных 1 и 0. Вдобавок булевский тип делает истинные значения более явными в коде. Скажем, бесконечный цикл теперь можно записать как `while True`: вместо менее понятного варианта `while 1:`. Аналогично появляется возможность более ясной инициализации флагов на подобие `flag = False`. В части III мы обсудим эти операторы более подробно.

Однако для большинства практических целей вы можете трактовать `True` и `False` так, как если бы они были заранее определенными переменными, установленными в целые значения 1 и 0. Большинство программистов в любом случае предварительно присваивали `True` и `False` значения 1 и 0; тип `bool` просто делает это стандартом.

Тем не менее, его реализация может привести к любопытным результатам. Так как `True` – целое число 1 со специальным форматом отображения, выражение `True + 4` в Python дает целое число 5!

```
>>> type(True)
<class 'bool'>
>>> isinstance(True, int)
True
>>> True == 1          # То же самое значение
True
>>> True is 1          # Но другой объект: см. следующую главу
False
>>> True or False     # То же что и 1 or 0
True
>>> True + 4           # M-да
5
```

Поскольку вы вряд ли встретите в реальном коде Python выражения, подобные последним, можете спокойно игнорировать любые его более глубокие метафизические последствия.

В главе 9 мы вернемся к булевским значениям для определения понятия истинности в Python и снова в главе 12, чтобы посмотреть, как работают булевские операции *вроде and и or*.

Численные расширения

Наконец, несмотря на то, что основные числовые типы Python обеспечивают достаточно мощь для большинства приложений, доступна крупная библиотека сторонних расширений с открытым кодом, которые предназначены для удовлетворения специализированных потребностей. Учитывая популярность численного программирования на Python, вы обнаружите обилие развитых инструментов.

Например, если вам необходимо заниматься серьезным перемалыванием чисел, то дополнительное расширение для Python под названием *NumPy* (Numeric Python) предоставляет расширенные инструменты численного программирования, такие как матричный тип данных, обработка векторов и библиотеки сложных вычислений. Группы, занимающиеся научными расчетами в местах вроде Лос-Аламос и НАСА, используют Python с расширением NumPy для решения задач, которые раньше решались с помощью C++, FORTRAN или Matlab. Комбинацию Python и NumPy часто считают похожей на бесплатную и более гибкую версию Matlab – вы получаете эффективность NumPy вместе с языком Python и его библиотеками.

Из-за высокой сложности NumPy в настоящей книге не рассматривается. В веб-сети вы без труда найдете дополнительную поддержку для численного программирования на Python, включая инструменты работы с графикой и диаграммами, типы с плавающей точкой расширенной точности, библиотеки обработки статистических данных и популярный пакет *SciPy*. Также имейте в виду, что в текущий момент NumPy является необязательным расширением; оно не поставляется вместе с Python и должно устанавливаться отдельно.

Резюме

В главе был проведен экскурс в типы числовых объектов Python и операции, которые к ним можно применять. Попутно мы исследовали стандартные целочисленные типы и типы с плавающей точкой, а также ряд более экзотических и менее распространенных типов наподобие комплексных чисел, десятичных чисел, дробей и множеств. Мы обсудили синтаксис выражений Python, преобразования между типами, побитовые операции и разнообразные лiteralные формы для записи чисел в сценариях.

Далее в этой части книги мы продолжим подробный обзор типов, восполняя недостающие детали об очередном типе объектов – строке. Однако в следующей главе мы займемся исследованием присваивания переменных. Оно представляет собой, пожалуй, наиболее фундаментальную идею в Python, поэтому обязательно ознакомьтесь со следующей главой, прежде чем двигаться дальше. А сейчас наступило время для традиционных контрольных вопросов.

Проверьте свои знания: контрольные вопросы

1. Каким будет значение выражения $2 * (3 + 4)$ в Python?
2. Каким будет значение выражения $2 * 3 + 4$ в Python?
3. Каким будет значение выражения $2 + 3 * 4$ в Python?
4. Какие инструменты можно использовать для нахождения квадратного корня и квадрата числа?
5. Каким будет тип результата выражения $1 + 2.0 + 3$?
6. Как можно усекать и округлять число с плавающей точкой?
7. Как можно преобразовывать целое число в число с плавающей точкой?
8. Как можно отображать целое число в восьмеричной, шестнадцатеричной или двоичной форме?
9. Как можно преобразовывать строку восьмеричных, шестнадцатеричных или двоичных цифр в простое целое число?

Проверьте свои знания: ответы

1. Значением выражения будет 14, результат $2 * 7$, поскольку круглые скобки приводят к выполнению сложения раньше умножения.
2. Значением выражения будет 10, результат $6 + 4$. В отсутствие круглых скобок применяются правила старшинства операций Python, а согласно табл. 5.2 умножение имеет более высокий приоритет, чем сложение.
3. Значением выражения будет 14, результат $2 + 12$, по тем же причинам старшинства, что и в предыдущем ответе.

4. Функции для получения квадратного корня, а также числа *pi*, тангенсов и многое другое доступны в импортируемом модуле `math`. Чтобы найти квадратный корень числа, импортируйте `math` и вызовите `math.sqrt(N)`. Для нахождения квадрата числа используйте либо выражение возвведения в степень `X ** 2`, либо встроенную функцию `pow(X, 2)`. Любой из двух последних приемов позволяет также вычислить квадратный корень в случае указания степени 0.5 (например, `X ** .5`).
5. Результатом будет число с плавающей точкой: целые числа преобразуются в числа с плавающей точкой (наиболее сложный тип в выражении) и для вычисления применяется математика с плавающей точкой.
6. Усечение производят функции `int(N)` и `math.trunc(N)`, а округление — функция `round(N, digits)`. Можно также округлять в меньшую сторону с помощью `math.floor(N)` и округлять для отображения посредством операций форматирования строк.
7. Функция `float(I)` преобразует целое число в число с плавающей точкой; смешивание целых чисел и чисел с плавающей точкой внутри выражения также приводит к преобразованию. В определенном смысле операция деления / из Python 3.X также производит преобразование — она всегда возвращает результат, который содержит остаток, даже если оба операнда являются целыми.
8. Встроенные функции `oct(I)` и `hex(I)` возвращают для целого числа строковые представления в восьмеричной и шестнадцатеричной формах. Вызов `bin(I)` возвращает строку с двоичными цифрами в Python 2.6, 3.0 и последующих версиях. Операция форматирования строк % и строковый метод `format` также позволяют выполнять подобные преобразования.
9. Для преобразования строк восьмеричных, шестнадцатеричных и двоичных цифр в нормальные целые числа можно использовать функцию `int(S, base)`, передавая ей в качестве основания системы счисления (`base`) соответственно 8, 16 и 2. Для той же самой цели можно применять и функцию `eval(S)`, но она сопряжена с более высокими затратами и проблемами в плане безопасности. Обратите внимание, что в памяти компьютера целые числа всегда хранятся в двоичной форме; преобразования происходят просто в целях отображения.

Кратко о динамической тиปизации

В предыдущей главе мы начали углубленное исследование основных типов объектов Python с изучения числовых типов Python и операций над ними. В следующей главе мы возобновим тур по типам объектов, но сначала важно, чтобы вы получили представление о том, что может считаться наиболее фундаментальной идеей в программировании на Python и определенно является базисом лаконичности и гибкости языка Python – динамической типизации и подразумеваемом ею полиморфизме.

Как вы увидите здесь и повсюду в книге, в Python мы не объявляем конкретные типы объектов, используемых в сценариях. На самом деле большинство программ не должны даже заботиться о конкретных типах; взамен иногда они становятся естественным образом применимыми в большем числе контекстов, чем планировалось заранее. Поскольку динамическая типизация является корнем такой гибкости, равно как и потенциальным камнем преткновения для новичков, давайте кратко ознакомимся с данной моделью.

Случай отсутствия операторов объявления

Если у вас есть опыт использования компилируемых либо статически типизируемых языков вроде C, C++ или Java, то в этом месте книги вы можете оказаться слегка сбиты с толку. До сих пор мы применяли переменные, не объявляя их существование или типы, и все как-то работало. Скажем, когда мы набираем `a = 3` в интерактивном сеансе или в файле программы, то каким образом Python узнает, что `a` должно означать целое число? В сущности, как Python узнает, что вообще собой представляет `a`?

Начав задаваться вопросами подобного рода, вы попадаете во владения модели *динамической типизации* Python. Типы в Python определяются автоматически во время выполнения, а не в ответ на объявления в коде. Это означает, что вы никогда не объявляете переменные заблаговременно (концепция, которую вероятно проще уловить, если постоянно помнить о том, что все сводится к переменным, объектам и связям между ними).

Переменные, объекты и ссылки

Во многих примерах, показанных до сих пор в книге, вы видели, что когда мы выполняем в Python оператор присваивания, такой как `a = 3`, то он работает, хотя мы никогда не сообщали Python об использовании имени `a` в качестве переменной или о том, что `a` обозначает объект целочисленного типа. В языке Python все складывается очень естественным образом, как описано далее.

Создание переменных

Переменная (также называемая в Python именем), подобная `a`, создается при первом присваивании ей значения в коде. Последующие присваивания изменяют значение уже созданного имени. Формально Python обнаруживает некоторые имена перед выполнением кода, но вы можете думать об этом так, как будто переменные создаются начальными присваиваниями.

Типы переменных

Переменная никогда не располагает какой-либо информацией о типе или о связанных с ним ограничениях. Понятие типа обитает в объектах, не в именах. Переменные являются обобщенными по своей природе; они всегда просто указываются на определенный объект в конкретный момент времени.

Использование переменных

Когда переменная встречается в выражении, она тотчас же заменяется объектом, на который в текущий момент ссылается, каким бы он ни был. Более того, все переменные должны быть присвоены, прежде чем их можно будет применять; ссылка на неприсвоенные переменные приводит к ошибкам.

В итоге переменные создаются, когда присваиваются, могут ссылаться на объекты любых типов и должны быть присвоены перед ссылкой на них. Это означает, что вам никогда не придется объявлять имена, используемые в сценарии, но вы обязаны инициализировать имена, прежде чем сможете обновлять их; например, счетчик должен быть инициализирован нулем до того, как можно будет его увеличивать.

Такая модель динамической типизации разительно отличается от модели типизации традиционных языков. В самом начале модель обычно легче понять, если четко осознать разницу между именами и объектами. Скажем, когда переменной присваивается значение:

```
>>> a = 3    # Присвоить имени объект
```

то, по крайней мере, концептуально Python произведет три отдельных шага, чтобы выполнить запрос. Следующие шаги отражают работу всех присваиваний в языке Python.

1. Создание объекта для представления значения 3.
2. Создание переменной `a`, если она еще не существует.
3. Связывание переменной `a` с новым объектом 3.

Совокупным результатом будет структура внутри Python, которая напоминает представленную на рис. 6.1. Как видно на рис. 6.1, переменные и объекты хранятся в разных местах памяти и связываются посредством ссылок (ссылка показана в виде стрелки). Переменные всегда указывают на объекты и никогда на другие переменные, но более крупные объекты могут быть связаны с другими объектами (например, объект списка имеет ссылки на содержащиеся в нем объекты).



Рис. 6.1. Имена (они же переменные) и объекты после выполнения присваивания $a = 3$. Переменная a становится ссылкой на объект 3. Внутренне переменная в действительности представляет собой указатель на область памяти объекта, созданного в результате выполненияliteralного выражения 3

Такие связи от переменных к объектам в Python называются *ссылками*; т.е. ссылка – это своего рода ассоциация, реализованная как указатель в памяти¹. Всякий раз, когда переменная используется в более позднее время (т.е. на нее производится ссылка), Python автоматически следует по связи от переменной к объекту. Конкретная ситуация проще, чем может вытекать из терминологии.

- Переменные – это записи в системной таблице, в которых предусмотрены места для связей с объектами.
- Объекты – это области выделенной памяти с достаточным пространством для представления значений, для которых они предназначены.
- Ссылки – это следуемые указатели от переменных к объектам.

По крайней мере, на понятийном уровне каждый раз, когда вы в своем сценарии генерируете новое значение путем выполнения выражения, Python создает новый *объект* (т.е. участок памяти) для представления этого значения. В целях оптимизации Python внутренне кеширует и повторно использует определенные виды неизменяемых объектов, такие как небольшие целые числа и строки (каждый 0 на самом деле не является новым участком памяти; мы рассмотрим поведение кеширования позже). Но с логической точки зрения все работает так, как будто значение результата каждого выражения представляет собой отдельный объект, а каждый объект – отдельный участок памяти.

Говоря формально, объекты имеют более сложную структуру, чем только пространство, достаточное для представления их значений. Каждый объект также содержит два стандартных заголовочных поля: *обозначение типа*, применяемое для пометки типа объекта, и *счетчик ссылок*, используемый для определения, когда можно освободить память, которую занимает объект. Чтобы понять, как указанные два заголовочных поля вписываются в модель, нам необходимо двигаться дальше.

¹ Читатели с опытом работы на языке C могут счесть ссылки Python похожими на указатели C (адреса в памяти). Фактически ссылки реализованы в виде указателей, и они часто исполняют те же самые роли, особенно с объектами, которые разрешают изменение на месте (позже мы обсудим сказанное более подробно). Однако поскольку в случае применения ссылки всегда подвергаются автоматическому разыменованию, вы никогда не сможете сделать что-то полезное с самой ссылкой; это свойство, которое устраняет обширную категорию ошибок, допускаемых в C. Но вы можете думать о ссылках Python как об указателях *void** в C, по которым происходит автоматическое следование всякий раз, когда они используются.

Типы обитают в объектах, не в переменных

Чтобы посмотреть, как объекты вступают в игру, проследим за происходящим во время многократного присваивания переменной:

```
>>> a = 3          # Переменная является целым числом
>>> a = 'spam'    # Теперь она стала строкой
>>> a = 1.23      # А теперь - числом с плавающей точкой
```

Код нельзя считать типичным для языка Python, но он работает — переменная `a` начинает свое существование как целое число, затем становится строкой и, наконец, числом с плавающей точкой. Особенno странным пример покажется бывшим программистам на C, поскольку он выглядит так, будто бы *тип* `a` изменяется с целочисленного на строковый, когда вводится `a = 'spam'`.

Тем не менее, на самом деле тут происходит кое-что другое. В Python все гораздо проще. *Имена* не имеют типов; как утверждалось ранее, типы обитают в объектах, не в именах. В предыдущем листинге мы лишь изменяем `a` для ссылки на другие объекты. Из-за того, что переменные не имеют типов, мы в действительности не изменяем тип переменной `a`; мы просто делаем переменную ссылкой на объект другого типа. Фактически снова мы можем сказать о переменной в Python лишь то, что она ссылается на определенный объект в конкретный момент времени.

С другой стороны, *объектам* известно, какого они типа — каждый объект содержит заголовочное поле, которое помечает объект его типом. Например, целочисленный объект `3` будет содержать значение `3` плюс обозначение, которое сообщает Python о том, что объект является целым числом (строго говоря, указатель на объект, называемый `int`, т.е. имя целочисленного типа). Обозначение типа строкового объекта `'spam'` взамен указывает на строковый тип (называемый `str`). Так как объектам известны свои типы, переменным знать их вовсе не обязательно.

Вспомните, что в Python типы ассоциированы с объектами, а не с переменными. В обычном коде заданная переменная, как правило, будет ссылаться лишь на один вид объекта. Но поскольку это не требование, вы обнаружите, что код Python оказывается гораздо более гибким, чем привычно для вас — если вы используете язык Python надлежащим образом, то код может работать со многими типами автоматически.

Выше упоминалось, что объекты имеют два заголовочных поля, обозначение типа и счетчик ссылок. Чтобы понять последний, необходимо двигаться дальше и бегло взглянуть, что происходит в конце жизни объекта.

Объекты подвергаются сборке мусора

В листинге из предыдущего раздела мы присваивали переменной `a` объекты разных типов в каждом операторе. Но что происходит со значением, на которое ранее ссылалась переменная, при ее повторном присваивании? Скажем, что случится с объектом `3` после выполнения следующих операторов?

```
>>> a = 3
>>> a = 'spam'
```

Ответ таков: в Python при присваивании имени нового объекта область памяти, занимаемая предыдущим объектом, освобождается в случае, если на него не ссылается любое другое имя или объект. Такое автоматическое освобождение памяти объектов известно как *сборка мусора*, и она значительно упрощает жизнь программистов на языках вроде Python, которые ее поддерживают.

В целях иллюстрации рассмотрим пример, где в каждом присваивании имя `x` устанавливается в отличающийся объект:

```
>>> x = 42
>>> x = 'shrubbery'    # Освободить память, занимаемую 42
# (если нет ссылок где-то еще)
>>> x = 3.1415        # Освободить память, занимаемую 'shrubbery'
>>> x = [1, 2, 3]      # Освободить память, занимаемую 3.1415
```

Во-первых, обратите внимание, что `x` каждый раз устанавливается в объект отличающегося типа. И снова, хотя все в действительности не так, эффект выглядит, как если бы тип `x` менялся с течением времени. Не забывайте, что в Python типы обитают в объектах, а не в именах. Поскольку имена – это лишь обобщенные ссылки на объекты, код подобного рода работает вполне естественным образом.

Во-вторых, обратите внимание, что ссылки на объекты по пути отбрасываются. Каждый раз, когда `x` присваивается новый объект, Python освобождает память, занятую предыдущим объектом. Например, после присваивания строки `'shrubbery'` память, которую занимает объект 42, немедленно освобождается (при условии, что на него больше нет ссылок) – т.е. область памяти объекта автоматически попадает в пул свободного пространства, чтобы быть задействованной каким-то будущим объектом.

Внутренне Python совершает такой ловкий трюк за счет ведения в каждом объекте счетчика, который отслеживает количество ссылок, в текущий момент указывающих на объект. Как только и точно когда этот счетчик уменьшается до нуля, область памяти объекта автоматически освобождается. В листинге выше мы предполагаем, что каждый раз, когда `x` присваивается новый объект, счетчик ссылок предыдущего объекта уменьшается до нуля, вызывая освобождение занимаемой им памяти.

Самое непосредственно ощутимое преимущество сборки мусора заключается в том, что она означает возможность свободного применения объектов без всякой необходимости в выделении или освобождении памяти в сценарии. Python будет очищать неиспользуемое пространство в ходе выполнения программы. На практике в результате устраняется значительный объем кода, связанного с управлением памятью, который требуется в языках более низкого уровня, таких как C и C++.

Подробнее о сборке мусора в Python

Говоря формально, сборка мусора в Python основана главным образом на *счетчиках ссылок*, как здесь описано; однако она также имеет компонент, который обнаруживает и своевременно освобождает пространство памяти, занимаемое объектами с *циклическими ссылками*. Если вы уверены, что ваш код не создает циклов, тогда можете отключить этот компонент, но по умолчанию он включен.

Циклические ссылки являются классической проблемой в сборщиках мусора, основанных на счетчиках ссылок. Поскольку ссылки реализованы как указатели, вполне возможно, что объект ссылается на самого себя или на другой объект, который ссылается на самого себя. Скажем, в упражнении 6 в конце части I и его решении в приложении показано, насколько легко создать цикл, внедряя в список ссылку на самого себя (например, `L.append(L)`). Такое же явление может произойти в присваиваниях атрибутов объектов, которые созданы из классов, определяемых пользователем. Несмотря на относительную редкость подобных случаев, с ними нужно обходиться особым образом, потому что счетчики ссылок в таких объектах никогда не уменьшаются до нуля.

Для получения дополнительных сведений о детекторе циклов Python загляните в описание модуля `gc` в руководстве по стандартной библиотеке Python. Лучшая новость здесь в том, что управление памятью, основанное на сборке мусора, реализовано в Python людьми, обладающими высокой квалификацией в данной области.

Также следует отметить, что описание сборщика мусора Python в настоящей главе применимо только к стандартному Python (*CPython*); альтернативные реализации, обсуждаемые в главе 2, такие как Jython, IronPython и PyPy, могут использовать отличающиеся схемы, хотя общий эффект аналогичен – неиспользуемое пространство памяти освобождается автоматически, но не всегда немедленно.

Разделяемые ссылки

Ранее мы видели, что происходит, когда одиночной переменной присваиваются ссылки на объекты. Давайте теперь введем в наше взаимодействие еще одну переменную и посмотрим, что случится с именами и объектами:

```
>>> a = 3  
>>> b = a
```

Картина, возникающая после набора показанных двух операторов, изображена на рис. 6.2. Вторая команда приводит к тому, что Python создает переменную `b`; переменная `a` здесь используется и не присваивается, поэтому она заменяется объектом, на который ссылается (3), а в `b` обеспечивается ссылка на данный объект. Совокупный эффект в том, что переменные `a` и `b` в итоге ссылаются на *тот же самый* объект (т.е. указывают на один и тот же участок памяти).

Такой сценарий в Python с множеством имен, ссылающихся на тот же самый объект, обычно называется *разделяемой ссылкой* (а иногда просто *разделяемым объектом*). Обратите внимание, что когда это происходит, имена `a` и `b` не связаны друг с другом напрямую; на самом деле в Python вообще нет способа связать одну переменную с другой. Наоборот, обе переменные указывают на один и тот же объект через свои ссылки.

Далее предположим, что мы расширяем сеанс еще одним оператором:

```
>>> a = 3  
>>> b = a  
>>> a = 'spam'
```

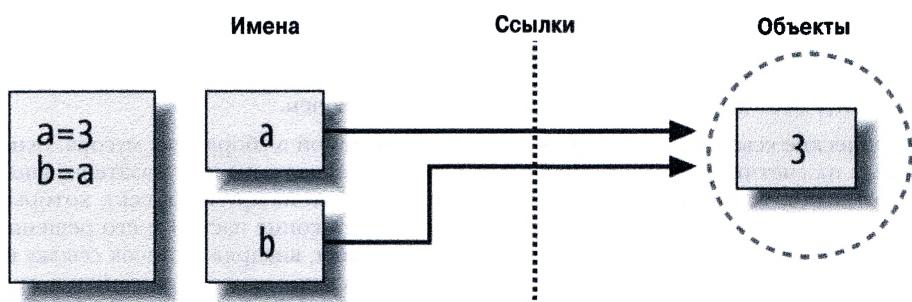


Рис. 6.2. Имена и объекты после выполнения следующего присваивания `b = a`. Переменная `b` становится ссылкой на объект 3. Внутренне переменная в действительности представляет собой указатель на область памяти объекта, созданного в результате выполнения лiteralного выражения 3

Как и со всеми присваиваниями Python, последний оператор просто создает новый объект для представления строкового значения 'spam' и устанавливает `a` в ссылку на этот новый объект. Тем не менее, он не изменяет значение `b`; переменная `b` по-прежнему ссылается на первоначальный объект, целое число 3. Результирующая структура представлена на рис. 6.3.

То же самое произошло бы в случае изменения `b` на 'spam' — присваивание модифицировало бы только `b`, но не `a`. Такое поведение также случается, если нет вообще никаких отличий между типами. Например, взгляните на следующие три оператора:

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

В приведенной последовательности возникают аналогичные события. Python создает переменную `a`, ссылающуюся на объект 3, и создает переменную `b`, ссылающуюся на тот же объект, что и `a` (см. рис. 6.2); как и ранее, последнее присваивание затем устанавливает `a` в совершенно другой объект (в данном случае целое число 5, которое является результатом выражения `+`). Оно не изменяет `b` в виде побочного эффекта. Фактически способы переписать значение объекта 3 вообще отсутствуют — как объяснялось в главе 4, целые числа неизменяемы и потому не могут быть модифицированы на месте.

Это можно представлять себе так, что в отличие от ряда языков переменные Python всегда являются указателями на объекты, а не метками допускающей изменение памяти: установка переменной в новое значение не изменяет первоначальный объект, но взамен приводит к ссылке переменной на совершенно другой объект. Совокупный эффект заключается в том, что само присваивание переменной влияет только на одну переменную, которой производится присваивание. Однако когда задействованы изменяемые объекты и изменения на месте, картина становится несколько иной; давайте проясним ситуацию.



Рис. 6.3. Имена и объекты после выполнения последнего присваивания `a = 'spam'`. Переменная `a` ссылается на новый объект (т.е. участок памяти), созданный в результате выполнения лiteralьного выражения 'spam', но переменная `b` все еще ссылается на первоначальный объект 3. Поскольку такое присваивание не является изменением на месте объекта 3, оно модифицирует только переменную `a`, но не `b`.

Разделяемые ссылки и изменения на месте

Как вы увидите позже в главах текущей части, существуют объекты и операции, которые вносят в объекты изменения на месте — *изменяемые* типы Python, включающие списки, словари и множества. Скажем, присваивание по смещению в списке фактически изменяет сам списоковый объект на месте, а не создает совершенно новый такой объект.

Несмотря на то что пока вы должны принять это на веру, такая отличительная особенность может быть очень важной в ваших программах. В отношении объектов, поддерживающих изменения на месте, вам необходимо больше заботиться о разделяемых ссылках, потому что модификация одного имени может повлиять на другие имена. В противном случае ваши объекты могут изменяться без видимой причины. Учитывая, что все присваивания основаны на ссылках (включая передачу аргументов функциям), подобное вполне возможно.

Для иллюстрации сказанного давайте еще раз обратимся к списковым объектам, введенным в главе 4. Вспомните, что списки, которые поддерживают присваивания по позиции на месте, представляют собой просто коллекции других объектов, записанные в квадратных скобках:

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
```

Здесь L1 — список, содержащий объекты 2, 3 и 4. Элементы внутри списка доступны по своим позициям, поэтому L1[0] ссылается на объект 2, т.е. первый элемент в списке L1. Разумеется, списки сами по себе являются объектами в точности как целые числа и строки. После выполнения показанных выше двух операторов L1 и L2 ссылаются на тот же самый разделяемый объект подобно a и b в предыдущем примере (см. рис. 6.2). А теперь, как и ранее, расширим взаимодействие следующим оператором:

```
>>> L1 = 24
```

Такое присваивание просто устанавливает L1 в другой объект; L2 по-прежнему ссылается на первоначальный список. Тем не менее, если мы слегка изменим синтаксис оператора, то получим совершенно другой результат:

```
>>> L1 = [2, 3, 4]          # Изменяемый объект
>>> L2 = L1                # Создает ссылку на тот же самый объект
>>> L1[0] = 24              # Изменение на месте
>>> L1                      # L1 отличается
[24, 3, 4]
>>> L2                      # Но отличается и L2!
[24, 3, 4]
```

На самом деле мы здесь не модифицировали сам список L1, а изменили компонент *объекта*, на который L1 ссылается. Изменение такого рода перезаписывает часть значения спискового объекта на месте. Однако поскольку списоковый объект совместно используется другой переменной (она ссылается на него), изменение на месте подобного вида оказывает влияние не только на L1. В итоге вы должны осознавать, что такие изменения могут воздействовать на другие части программы. В приведенном примере эффект модификации отражается и в списке L2, т.к. он ссылается на тот же самый объект, что и L1. В действительности мы не изменили список L2, но его значение стало другим, потому что он ссылается на объект, который был перезаписан на месте.

Такое поведение характерно только для изменяемых объектов, которые поддерживают изменения на месте, и обычно является желательным, но вы обязаны знать, как оно работает, и ожидать его. Кроме того, так происходит по умолчанию: если вас не

устраивает это поведение, тогда можете запросить у Python *копирование* объектов вместе с ссылки на них. Доступны разнообразные способы копирования списка, включая применение встроенной функции `list` и стандартного библиотечного модуля `copy`. Пожалуй, наиболее распространенный способ предусматривает нарязание с начала до конца (нарезание подробно обсуждается в главах 4 и 7):

```
>>> L1 = [2, 3, 4]
>>> L2 = L1[:]          # Создать копию L1 (или list(L1), copy.copy(L1) и т.д.)
>>> L1[0] = 24
>>> L1
[24, 3, 4]
>>> L2              # L2 не изменяется
[2, 3, 4]
```

Здесь изменение, внесенное через `L1`, не влияет на `L2`, поскольку `L2` ссылается на копию, а не на оригинал объекта, на который ссылается `L1`; т.е. две переменные указывают на разные участки памяти.

Обратите внимание, что такой подход с нарязанием не будет работать с другими изменяемыми основными типами, словарями и множествами, потому что они не являются последовательностями. Для копирования словарей или множеств взамен нужно использовать вызов их метода `X.copy()` (начиная с версии Python 3.3, списки тоже его имеют) или передавать исходный объект их именам типов, `dict` и `set`. Вдобавок следует отметить, что стандартный библиотечный модуль `copy` имеет вызов для копирования объекта любого типа обобщенным образом и вызов для копирования структур с вложенными объектами, например, словаря с вложенными списками:

```
import copy
X = copy.copy(Y)      # Создать "поверхностную" копию верхнего уровня
                      # любого объекта Y
X = copy.deepcopy(Y) # Создать глубокую копию любого объекта Y:
                      # копировать все вложенные части
```

В главах 8 и 9 мы более глубоко исследуем списки и словари, а также возвратимся к концепции разделяемых ссылок и копий. Пока имейте в виду, что объекты, которые могут изменяться на месте (т.е. изменяемые объекты), всегда открыты для эффектов подобного рода в любом коде, через который они проходят. В Python к ним относятся списки, словари, множества и ряд объектов, определенных с помощью операторов `class`. Если это не является желаемым поведением, тогда можете просто копировать объекты по мере необходимости.

Разделяемые ссылки и равенство

Ради полного раскрытия следует отметить, что описанное ранее в главе поведение сборки мусора для определенных типов может оказаться более сложным. Взгляните на приведенные далее операторы:

```
>>> x = 42
>>> x = 'shrubbery' # Память, выделенная под объект 42, теперь освобождена?
```

Поскольку Python кеширует и повторно использует небольшие целые числа и строки, как упоминалось ранее, память для объекта 42 возможно не освобождается здесь буквально; взамен объект вероятно останется в системной таблице для повторного применения в следующий раз, когда в коде появится 42. Тем не менее, память большинства объектов освобождается немедленно, как только на них перестают ссылаться; что касается тех, память которых не освобождается, механизм кеширования от-

ношения к ним не имеет. Скажем, из-за особенностей модели ссылок Python есть два разных способа проверки на равенство в программе Python. В целях демонстрации создадим разделяемую ссылку:

```
>>> L = [1, 2, 3]
>>> M = L          # M и L ссылаются на один и тот же объект
>>> L == M        # Одинаковые значения
True
>>> L is M        # Одинаковые объекты
True
```

Первый прием, операция `==`, предусматривает проверку, имеют ли два ссылаемых объекта одинаковые *значения*; такой метод почти всегда используется для проверок на предмет равенства в Python. Второй прием, операция `is`, проверяет *идентичность* — она возвращает `True`, только если оба имени указывают на точно тот же самый объект, поэтому операция `is` является гораздо более строгой формой проверки равенства и редко применяется в большинстве программ.

На самом деле операция `is` просто сравнивает указатели, которые реализуют ссылки, и при необходимости служит способом обнаружения разделяемых ссылок в коде. Она возвращает `False`, если имена указывают на эквивалентные, но разные объекты, как в случае выполнения двух разных лiteralных выражений:

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]      # M и L ссылаются на разные объекты
>>> L == M            # Одинаковые значения
True
>>> L is M           # Разные объекты
False
```

А теперь посмотрим, что произойдет, когда мы выполним те же операции на небольших числах:

```
>>> X = 42
>>> Y = 42          # Должны быть двумя разными объектами
>>> X == Y
True
>>> X is Y         # Как бы то ни было, тот же самый объект: кеширование в действии!
True
```

В последнем взаимодействии `X` и `Y` должны давать `True` в операции `==` (то же самое значение), но не в операции `is` (тот же самый объект), потому что мы выполняли два разных лiteralных выражения (42). Однако поскольку небольшие целые числа и строки кешируются и используются повторно, операция `is` сообщает о том, что они ссылаются на тот же самый одиночный объект.

Если вы действительно хотите заглянуть “за кулисы”, то всегда можете поинтересоваться у Python, сколько ссылок на объект имеется: функция `getrefcount` из стандартного модуля `sys` возвращает счетчик ссылок объекта. Скажем, когда я запрашиваю счетчик ссылок целочисленного объекта 1 в графическом пользовательском интерфейсе IDLE, то получаю отчет о его 647 повторных применениях (большинство которых относится к системному коду IDLE, а не моему); тем не менее, за пределами IDLE возвращается информация о 173 повторных использованиях, так что Python и сам накапливает объекты 1):

```
>>> import sys
>>> sys.getrefcount(1)    # 647 указателя на этот разделяемый участок памяти
647
```

Данный объект кешируется и его повторное применение не имеет отношения к вашему коду (если только вы не выполняете проверку `is!`). Из-за невозможности модификации неизменяемых чисел и строк на месте совершенно не играет роли, сколько ссылок имеется на тот же самый объект — каждая ссылка всегда будет видеть одно и то же неизмененное значение. Однако такое поведение отражает один из многих способов, которыми Python оптимизирует свою модель для обеспечения высокой скорости выполнения.

Динамическая типизация вездесуща

Конечно, на самом деле вам не придется рисовать диаграммы имен/объектов с окружностями и стрелками, чтобы использовать Python. Тем не менее, когда вы только начинаете программировать на Python, понять необычные случаи иногда помогает отслеживание структур их ссылок, как мы делали здесь. Например, если изменяемый объект модифицируется при передаче внутри программы, то вполне вероятно, что вы воочию сталкиваетесь с тем, о чём шла речь в настоящей главе.

Более того, даже если на этом этапе динамическая типизация кажется немного абстрактной, то со временем вы наверняка уладите вопрос. Поскольку в Python, похоже, *абсолютно все взаимодействует* через присваивание и ссылки, базовое понимание такой модели полезно во многих контекстах. Как вы увидите, динамическая типизация работает одинаково в операторах присваивания, аргументах функций, переменных циклов `for`, импортах модулей, атрибутах классов и т.д. Хорошая новость заключается в том, что в Python существует только *одна* модель присваивания; однажды овладев динамической типизацией, вы обнаружите, что она работает одинаково повсюду в языке.

На утилитарном уровне динамическая типизация означает для вас необходимость написания меньшего объема кода. Однако в равной степени важно и то, что динамическая типизация является корнем *полиморфизма* Python — концепции, которая была представлена в главе 4, и еще будет обсуждаться позже в книге.

Поскольку мы не ограничиваем типы в коде Python, он оказывается лаконичным и чрезвычайно гибким. Как вы увидите, при правильном применении динамическая типизация и вытекающий из нее полиморфизм производят код, который автоматически адаптируется к новым требованиям по мере развития ваших систем.

“Слабые” ссылки

В мире Python вы можете порой встречать термин “слабые ссылки” — довольно развитый инструмент, который связан с исследованной здесь моделью ссылок и подобно операции `is` без нее не может быть по-настоящему понят.

Вкратце слабая ссылка, реализуемая стандартным библиотечным модулем `weakref`, представляет собой ссылку на объект, которая сама по себе не препятствует выполнению сборки мусора в отношении этого объекта. Если последними оставшимися ссылками на объект оказываются слабые ссылки, тогда выделенная под объект память освобождается, а слабые ссылки автоматически удаляются (или уведомляются об освобождении как-то иначе).

Слабые ссылки могут быть удобными, скажем, в кешах для крупных объектов на основе словарей; без них одна лишь ссылка кеша приводила бы к хранению объекта в памяти бесконечно долго. Тем не менее, в действительности это просто специализированное расширение модели ссылок. За дополнительными сведениями обращайтесь в руководство по библиотеке Python.

Резюме

В главе мы более подробно рассмотрели модель динамической типизации Python – т.е. способ, которым Python автоматически отслеживает типы объектов, не требуя от нас написания операторов объявлений в сценариях. Попутно вы узнали, каким образом переменные и объекты связываются ссылками в Python; также обсуждалась идея сборки мусора, было показано, как разделяемые ссылки могут влиять на множество переменных, и приведены объяснения воздействия ссылок на понятие равенства в Python.

Поскольку в Python имеется лишь одна модель присваивания и оттого, что присваивание вездесуще в языке, важно понять данную модель, прежде чем двигаться дальше. Следующие контрольные вопросы должны помочь вам закрепить некоторые идеи, изложенные в главе. Затем мы возобновим наш тур по основным объектам, перейдя к рассмотрению строк.

Проверьте свои знания: контрольные вопросы

1. Взгляните на следующие три оператора. Изменяют ли они значение, выводимое для A?

```
A = "spam"
B = A
B = "shrubbery"
```
2. Взгляните на приведенные ниже три оператора. Изменяют ли они значение, выводимое для A?

```
A = ["spam"]
B = A
B[0] = "shrubbery"
```
3. Как насчет показанных далее трех операторов – изменяется ли A теперь?

```
A = ["spam"]
B = A[:]
B[0] = "shrubbery"
```

Проверьте свои знания: ответы

1. Нет: для A по-прежнему выводится "spam". Когда B присваивается строка "shrubbery", все что происходит – переменная B переустанавливается для указания на новый строковый объект. А и B изначально разделяются, т.е. указывают на тот же самый одиночный строковый объект "spam", но в Python два имени никогда не связываются вместе. Таким образом, установка B в другой объект не оказывает влияния на A. Кстати, то же самое справедливо, если бы последним оператором здесь был B = B + 'shrubbery' – конкатенация привела бы к созданию в качестве своего результата нового объекта, который затем был бы присвоен только B. Мы не можем перезаписывать строку (либо число или кортеж) на месте, потому что строки неизменяемы.
2. Да: теперь для A выводится ["shrubbery"]. Формально мы на самом деле не изменяем ни A, ни B; замен мы модифицируем часть объекта, на который ссылаются обе переменные (указывают на него), перезаписывая этот объект на месте через переменную B. Поскольку A ссылается на тот же самый объект, что и B, обновление отражается также в A.
3. Нет: для A по-прежнему выводится ["spam"]. Присваивание на месте через B на этот раз не оказывает воздействия, т.к. выражение среза создает копию спискового объекта до его присваивания переменной B. После второго оператора присваивания существуют два разных списковых объекта, которые имеют одинаковые значения (в Python они дают True при выполнении операции ==, но не is). Третий оператор изменяет значение спискового объекта, на который указывает B, но не того, на который указывает A.

Фундаментальные основы строк

До сих пор мы изучали числа и исследовали модель динамической типизации Python. Следующим важным типом в нашем углубленном туре по основным объектам Python будет *строка* – упорядоченная коллекция символов, используемая для хранения и представления текстовой и байтовой информации. Мы кратко затронули тему строк в главе 4. Здесь мы начнем их рассматривать более глубоко, заполнив детали, которые ранее были оставлены без внимания.

Вопросы, раскрываемые в главе

Прежде чем приступить, я хочу прояснить, что в главе *не* будет раскрываться. В главе 4 приводился краткий обзор строк *Unicode* и файлов – инструментов для работы с текстом, отличающимся от ASCII. Кодировка Unicode является ключевым инструментом для ряда программистов, в особенности тех, кто имеет дело с областью, связанной с Интернетом. Например, Unicode встречается на веб-страницах, в содержимом и заголовках электронной почты, передачах по протоколу FTP, в API-интерфейсах для построения графических пользовательских интерфейсов, в инструментах для работы с каталогами, а также в тексте HTML, XML и JSON.

В то же время Unicode может быть трудной темой для начинающих программистов, и многие (или большинство) программистов на Python, с которыми мне приходится встречаться в наши дни, все еще делают свою работу, пребывая в блаженном неведении о данной теме. В свете этого большая часть истории, касающейся Unicode, откладывается до главы 37, входящей в ту часть книги, которая посвящена расширенным темам и необязательна для чтения. Здесь мы сосредоточим внимание на основах строк.

Таким образом, в главе сообщается только часть истории, связанной со строками Python – та часть, которая применяется в большинстве сценариев и которую должны знать почти все программисты. В ней исследуется фундаментальный строковый тип `str`, который поддерживает текст ASCII и работает одинаково вне зависимости от используемой версии Python. Невзирая на преднамеренно ограниченный охват, поскольку `str` также поддерживает Unicode в Python 3.X, а отдельный тип `unicode` работает почти идентично типу `str` в Python 2.X, все изученное здесь будет также применимо напрямую к обработке Unicode.

Unicode: краткая история

Для читателей, которым небезразличен Unicode, я хотел бы предложить краткую сводку по его влиянию и указания по дальнейшему изучению. С формальной точки зрения ASCII – это простая форма текста Unicode, но лишь одна из многих возможных кодировок и алфавитов. В тексте из неанглоязычных источников могут использоваться совершенно отличающиеся буквы, а сам текст может кодироваться по-другому при хранении в файлах.

Как вы узнали в главе 4, проблема решается в Python путем проведения различий между текстовыми и двоичными данными, с отдельными типами строковых объектов и файловыми интерфейсами для каждого. Такая поддержка варьируется в зависимости от линейки Python.

- В Python 3.X существуют три строковых типа: `str` применяется для текста Unicode (в том числе ASCII), `bytes` используется для двоичных данных (включая закодированный текст), а `bytearray` является изменяемым вариантом типа `bytes`. Файлы работают в двух режимах: текстовом, который представляет содержимое как тип `str` и реализует кодировки Unicode, и двоичном, который имеет дело с низкоуровневым типом `bytes` и не выполняет какую-либо трансляцию данных.
- В Python 2.X строки `unicode` представляют текст Unicode, строки `str` поддерживают 8-битный текст и двоичные данные, а тип `bytearray` доступен в Python 2.6 и последующих версиях в результате обратного переноса из Python 3.X. Содержимое нормальных файлов – это просто байты, представленные как `str`, но модуль `codecs` позволяет открывать текстовые файлы Unicode, поддерживающий кодировки и представляет содержимое в виде объектов `unicode`.

Несмотря на такие отличия между версиями, если вам нужно позаботиться о кодировке Unicode, то вы обнаружите, что требуется относительно небольшое расширение – когда текст находится в памяти, он является строкой символов Python, которая поддерживает все основы, излагаемые в текущей главе. Фактически главное отличие Unicode часто лежит в шаге *трансляции* (также известной как *кодирование*), который требуется при перемещении информации в файлы и из них. Остальное по большому счету – всего лишь обработка строк.

Однако снова из-за того, что большинству программистов не нужно вникать в детали Unicode заблаговременно, почти все подробные сведения вынесены в главу 37. Когда вы будете готовы изучать эти более сложные концепции строк, я рекомендую просмотреть их предварительный обзор в главе 4 и полное раскрытие строк Unicode и байтовых строк в главе 37 после чтения материала, посвященного фундаментальным основам строк в настоящей главе.

Внимание в этой главе сосредоточено на базовом строковом типе и его операциях. Вы обнаружите, что рассматриваемые здесь приемы напрямую применимы также и к более сложным строковым типам в инструментальном наборе Python.

Основы строк

С функциональной точки зрения строки допускается использовать для представления почти всего, что может кодироваться как текст или байты. В отношении текста сюда входят символы и слова (скажем, ваше имя), содержимое текстовых файлов, загруженное в память, адреса в Интернете, исходный код Python и т.д. Строки могут

также применяться для хранения низкоуровневых байтов, используемых в файлах медиаданных и при передаче через сеть, а также кодированных и декодированных форм отличающегося от ASCII текста Unicode, который применяется в интернационализированных программах.

Вы также могли использовать строки в других языках. Строки Python исполняют ту же самую роль, что и массивы символов в языках вроде C, но по сравнению с массивами они кое в чем являются инструментом более высокого уровня. В отличие от C строки в Python сопровождаются мощным набором инструментов обработки. Кроме того, в отличие от языков, подобных C, в Python не предусмотрен отдельный тип для индивидуальных символов; взамен применяются односимвольные строки.

Строго говоря, строки Python относятся к категории *неизменяемых последовательностей*, а это означает, что содержащиеся в них символы имеют позиционный порядок слева направо и не могут быть модифицированы на месте. На самом деле строки являются первым представителем более крупного класса объектов, называемых *последовательностями*, которые мы будем изучать в книге. Уделите особое внимание операциям над последовательностями, введенным в главе, потому что они будут работать с другими типами последовательностей, которые мы рассмотрим позже, такими как списки и кортежи.

В табл. 7.1 приведен обзор распространенных строковых литералов и операций, обсуждаемых в главе. Пустая строка записывается в виде пары кавычек (одинарных или двойных), между которыми ничего нет, и существуют разнообразные способы записи строки в коде. В плане обработки строки поддерживают операции *выражений*, такие как конкатенация (объединение строк), нарезание (извлечение частей), индексация (извлечение по смещению) и т.д. Помимо операций выражений Python также предоставляет набор строковых *методов*, которые реализуют общие задачи, специфичные для строк, и *модули* для решения более сложных задач вроде сопоставления с образцом. Мы исследуем все названное далее в главе.

Таблица 7.1. Распространенные строковые литералы и операции

Операция	Описание
S = ''	Пустая строка
S = "spam's"	Двойные кавычки; то же самое, что и одинарные
S = 's\n\t\\x00m'	Управляющие последовательности
S = """... много строк ... """	Блочные строки в утроенных кавычках
S = r'\temp\spam'	Неформатированные строки (без управляющих символов)
B = b'sp\xc4m'	Байтовые строки в Python 2.6, 2.7 и 3.X (см. главы 4 и 37)
U = u'sp\u00c4m'	Строки Unicode в Python 2.X и 3.3+ (см. главы 4 и 37)
S1 + S2	Конкатенация, повторение
S*3	
S[i]	Индекс, срез, длина
S[i:j]	
len(S)	
"a %s parrot" % kind	Выражение форматирования строки
"a {0} parrot"\n .format(kind).find('pa')	Метод форматирования строки в Python 2.6, 2.7 и 3.X

Операция	Описание
S.find('pa') S.rstrip() S.replace('pa', 'xx') S.split(',') S.isdigit() S.lower() S.endswith('spam') 'spam'.join(strlist) S.encode('latin-1') B.decode('utf8')	Строковые методы (всего их 43): поиск, удаление пробельных символов, замена, разбиение по разделителю, проверка содержимого, преобразование регистра символов, проверка конца, объединение с разделителем, кодирование Unicode, декодирование Unicode и т.д. (табл. 7.3)
for x in S: print(x) 'spam' in S [c * 2 for c in S] map(ord, S)	Итерация, членство
re.match('sp(.*?)am', line)	Сопоставление с образцом: библиотечный модуль

Кроме основного набора строковых инструментов, перечисленных в табл. 7.1, в Python также поддерживается более развитая обработка строк на основе образцов с помощью стандартного библиотечного модуля `re` (от “regular expression” – “регулярные выражения”), представленного в главах 4 и 37, и инструменты обработки текста даже еще более высокого уровня, такие как анализаторы XML (кратко обсуждаются в главе 37). Тем не менее, книга ориентирована на фундаментальные инструменты, перечисленные в табл. 7.1.

Чтобы раскрыть основы, глава начинается с обзора форм строковых литералов и выражений, после чего рассматриваются более сложные инструменты вроде строковых методов и форматирования. В состав Python входят многие строковые инструменты, и мы не сможем описать их все в книге; для этого предназначено руководство по библиотеке Python и справочники. Наша цель – исследовать достаточно распространенные инструменты, чтобы обеспечить репрезентативную выборку; например, методы, которые здесь не демонстрируются в действии, почти полностью аналогичны тем, которые будут проиллюстрированы далее в главе.

Строковые литералы

В общем и целом строки в Python использовать довольно легко. Самой сложной связанный с ними вещью следует считать, пожалуй, наличие слишком многих способов их записи в коде:

- одинарные кавычки – `'spa"m'`;
- двойные кавычки – `"spa'm"`;
- утроенные кавычки – `'''... spam ...'''`, `"""... spam ..."""`;
- управляемые последовательности – `"s\t\r\na\0m"`;
- неформатированные строки – `r"C:\new\test.spm"`;
- байтовые литералы в Python 3.X и 2.6+ (см. главы 4 и 37) – `b'sp\x01am'`;
- литералы Unicode в Python 2.X и 3.3+ (см. главы 4 и 37) – `u'eggs\u0020spam'`.

Формы с одинарными и двойными кавычками безоговорочно являются наиболее распространенными; другие исполняют специализированные роли, и дальнейшее обсуждение последних двух расширенных форм мы отложим до главы 37. Давайте кратко взглянем на все остальные варианты по очереди.

Строки в одинарных и двойных кавычках являются одинаковыми

Среди строк символы одинарных и двойных кавычек взаимозаменяемы. То есть строковые литералы можно записывать заключенными либо в две одинарных, либо в две двойных кавычки — две формы работают одинаково и возвращают объект того же самого типа. Например, следующие две строки идентичны:

```
>>> 'shrubbery', "shrubbery"  
('shrubbery', 'shrubbery')
```

Причина поддержки обеих форм связана с тем, что она позволяет внедрять символ кавычки другого вида внутрь строки, не отменяя его с помощью обратной косой черты. Вы можете внедрять символ одинарной кавычки в строку, заключенную в символы двойной кавычки, и наоборот:

```
>>> 'knight"s', "knight's"  
('knight"s', "knight's")
```

В книге в целом предпочтение отдается применению *одинарных* кавычек вокруг строк, т.к. их чуть легче читать, кроме случаев внедрения одинарной кавычки в строку. Это чисто субъективный выбор стиля, но Python отображает строки аналогичным способом и так в наши дни поступает большинство программистов на Python, а потому вероятно вы должны делать то же самое.

Обратите внимание, что запятая здесь важна. Без нее Python выполняет *автоматическую конкатенацию* соседних строк в любом выражении, хотя почти так же просто поместить между ними операцию +, чтобы инициировать конкатенацию явно (как будет показано в главе 12, помещение такой формы в круглые скобки делает возможным разнесение на несколько строк):

```
>>> title = "Meaning " 'of' " Life"      # Неявная конкатенация  
>>> title  
'Meaning of Life'
```

Добавление запятых между этими строками привело бы в результате к созданию кортежа, а не строки. Кроме того, Python везде выводит строки в одинарных кавычках, если только они не были внедрены. При необходимости вы можете также внедрять символы кавычек, отменяя их посредством обратной косой черты:

```
>>> 'knight\'s', "knight\"s"  
("knight's", 'knight"s')
```

Чтобы понять, почему, нужно знать, как вообще работает отмена.

Управляющие последовательности представляют специальные символы

В последнем примере кавычка внедряется внутрь строки путем ее предварения обратной косой чертой. Это представляет общий шаблон в строках: обратная косая черта используется для введения специальных кодировок символов, известных как *управляющие последовательности*.

Управляющие последовательности позволяют внедрять в строки символы, которые не могут быть легко набраны на клавиатуре. Символ \ и один или более следующих за ним символов в строковом литерале в результирующем строковом объекте заменяются *одиночным* символом, который имеет двоичное значение, указанное управляющей последовательностью. Скажем, вот пятисимвольная строка, в которую внедрены символы новой строки и табуляции:

```
>>> s = 'a\nb\tc'
```

Два символа \n обозначают одиночный символ – двоичное значение символа новой строки в таблице символов (код символа 10 в ASCII). Подобным образом последовательность \t заменяется символом табуляции. То, как строка будет выглядеть при выводе, зависит от способа ее вывода. Интерактивный автоматический вывод показывает специальные символы как управляющие последовательности, но print их интерпретирует:

```
>>> s  
'a\nb\tc'  
>>> print(s)  
a  
b      c
```

Чтобы выяснить, сколько действительных символов содержится в строке, примите встроенную функцию len, которая возвращает фактическое количество символов в строке независимо от того, каким образом они кодируются или отображаются:

```
>>> len(s)  
5
```

Строка имеет длину пять символов: она содержит ASCII-символ *a*, символ новой строки, ASCII-символ *b* и т.д.



Если вы привыкли к тексту, который весь представлен в кодировке ASCII, то заманчиво думать, что этот результат означает также 5 *байтов*, но поступать так не следует. На самом деле “байты” в мире Unicode ничего не значат. Прежде всего, строковый объект в Python, вероятно, занимает больше памяти.

Что более важно, содержимое и длина строки отражают *кодовые точки* (идентифицирующие числа) в терминах Unicode, где одиночный символ не обязательно отображается напрямую на один байт, либо когда кодируется в файлах, либо когда хранится в памяти. Такое отображение может оставаться правильным для простого текста в 7-битной кодировке ASCII, но даже это зависит как от типа внешней кодировки, так и от используемой схемы внутреннего хранилища. Например, в UTF-16 символы ASCII занимают несколько байтов в файлах и могут требовать 1, 2 или 4 байта в памяти в зависимости от того, каким образом Python выделяет для них пространство. Для текста, отличающегося от ASCII, чьи значения символов могут быть слишком большими, чтобы уместиться в 8-битный байт, отображение символов на байты вообще неприменимо.

Фактически ради ясности в Python 3.X строки str формально определены как *последовательности кодовых точек Unicode*, но не байтов. При желании можете почитать главу 37, в которой рассказывается, как строки хранятся внутренне. Пока безопаснее всего думать о *символах*, а не *байтах* в строках. Доверьтесь мне; как бывшему программисту на С мне тоже пришлось избавиться от этой привычки!

Обратите внимание, что исходные символы обратной косой черты в предыдущем результате в действительности не хранятся со строкой в памяти; они используются лишь для описания значений специальных символов, подлежащих сохранению в строке. Для кодирования таких специальных символов Python распознает полный набор управляющих последовательностей, приведенный в табл. 7.2.

Таблица 7.2. Символы, представляемые в строке посредством обратной косой черты

Управляющая последовательность	Описание
\новая строка	Игнорируется (строка продолжения)
\ \	Обратная косая черта (сохраняет один символ \)
\ '	Одинарная кавычка (сохраняет ')
\ "	Двойная кавычка (сохраняет ")
\a	Звонок
\b	Забой
\f	Перевод страницы
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\xhh	Символ с шестнадцатеричным значением hh (точно 2 цифры)
\ooo	Символ с восьмеричным значением ooo (до 3 цифр)
\0	Пустой: двоичный символ 0 (не конец строки)
\N{ идентификатор }	Идентификатор базы данных Unicode
\uhhhh	Символ Unicode с 16-битным шестнадцатеричным значением
\Uhhhhhh	Символ Unicode с 32-битным шестнадцатеричным значением ^a
\остальное	Не управляющая последовательность (сохраняет \ и остальное)

^a Управляющая последовательность \Uhhhh... занимает в точности восемь шестнадцатеричных цифр (h); в Python 2.X последовательности \u и \U распознаются только в строковых литералах Unicode, но в Python 3.X могут применяться в нормальных строках (которые *представлены* в Unicode). В *байтовых* литералах Python 3.X шестнадцатеричные и восьмеричные управляющие последовательности обозначают байт с заданным значением; в *строковых* литералах такие управляющие последовательности обозначают символ Unicode с заданным значением кодовой точки. Дополнительные сведения об управляющих последовательностях Unicode ищите в главе 37.

Некоторые управляющие последовательности позволяют встраивать абсолютные двоичные значения в символы строки. Скажем, вот пятысимвольная строка, в которую встраиваются два символа с нулевыми двоичными значениями (представленные как восьмеричные управляющие последовательности):

```
>>> s = 'a\0b\0c'  
>>> s  
'a\x00b\x00c'  
>>> len(s)  
5
```

В Python нулевой (пустой) символ такого рода не завершает строку, как “нулевой байт” обычно делает в языке С. Взамен Python хранит в памяти и длину, и текст строки. На самом деле в Python *не предусмотрена* какой-либо символ для завершения строк. Ниже приведена строка, состоящая целиком из абсолютных двоичных управляемых кодов – двоичных значений 1 и 2 (представленных в восьмеричной форме), за которыми следует двоичное значение 3 (представленное в шестнадцатеричном виде):

```
>>> s = '\001\002\x03'  
>>> s  
'\x01\x02\x03'  
>>> len(s)  
3
```

Обратите внимание, что Python отображает непечатаемые символы в шестнадцатеричной форме независимо от того, как они были указаны. Вы можете свободно комбинировать абсолютные управляющие коды и символические управляющие последовательности из табл. 7/2. Следующая строка содержит символы “spam”, табуляцию и новую строку, а также символ с абсолютным нулевым значением, представленный в шестнадцатеричном виде:

```
>>> S = "s\tp\na\x00m"
>>> S
's\tp\na\x00m'
>>> len(S)
7
>>> print(S)
s      p
a m
```

Знание всего этого становится более важным при обработке файлов двоичных данных в Python. Поскольку в сценариях их содержимое представляется как строки, вполне нормально обрабатывать двоичные файлы, которые содержат любые двоичные значения байтов – при открытии в двоичном режиме файлы возвращают строки низкоуровневых байтов (больше сведений о файлах можно почерпнуть из глав 4, 9 и главы 37 второго тома).

Наконец, как вытекает из последней записи в табл. 7.2, если Python не распознает символ после \ в качестве допустимого управляющего кода, то он просто оставляет обратную косую черту в результирующей строке:

```
>>> x = "C:\py\code"      # Сохраняет \ буквально (и отображает как \\)
>>> x
'C:\\py\\code'
>>> len(x)
10
```

Однако если только вы не обладаете способностью запомнить всю табл. 7.2 (безусловно, нейроны можно использовать и эффективнее), то вероятно не должны полагаться на такое поведение. Чтобы явно представить буквальную обратную косую черту в строке, удваивайте ее (\\" для \) или применяйте неформатированные строки, как объясняется в следующем разделе.

Неформатированные строки подавляют управляемые последовательности

Как уже выяснилось, управляемые последовательности удобны для внедрения кодов специальных символов внутрь строк. Тем не менее, иногда специальная трактовка обратной косой черты для введения управляемой последовательности может вызывать затруднения. Скажем, удивительно часто новички в Python пытаются открыть файл, передавая в качестве аргумента имя файла, которое выглядит примерно так:

```
myfile = open('C:\new\text.dat', 'w')
```

Они полагают, что откроют файл по имени `text.dat` в каталоге `C:\new`. Проблема здесь в том, что `\n` обозначает символ новой строки, а `\t` заменяется табуляцией. В действительности такой вызов `open` пытается открыть файл с именем `C: (новая строка) ew (табуляция) ext.dat`, приводя к не особенно выдающимся результатам.

Как раз в таких ситуациях становятся полезными неформатированные строки. Если прямо перед открывающей кавычкой строки указать букву `r` (в нижнем или верхнем регистре), тогда механизм отмены отключается. В итоге Python сохраняет символы обратной косой черты буквально, в точности как они набраны. Следовательно, для устранения проблемы с именем файла в Windows нужно просто не забыть добавить букву `r`:

```
myfile = open(r'C:\new\text.dat', 'w')
```

В качестве альтернативы, поскольку две обратные косые черты на самом деле представляют собой управляемую последовательность для одной обратной косой черты, предохранить обратные косые черты в имени файла можно, просто продублировав их:

```
myfile = open('C:\\new\\text.dat', 'w')
```

Фактически сам Python временами использует такую схему дублирования, когда выводит строки с внедренными обратными косыми чертами:

```
>>> path = r'C:\\new\\text.dat'
>>> path                      # Отображает в формате, как в коде Python
'C:\\new\\text.dat'
>>> print(path)               # Отображает в формате, дружественном к пользователю
C:\\new\\text.dat
>>> len(path)                 # Длина строки
15
```

Как и в случае числового представления, стандартный формат в интерактивной подсказке предусматривает отображение результатов в виде, который они имели бы, находясь в коде, а потому отменяет обратные косые черты в выводе. Оператор `print` поддерживает более дружественный к пользователю формат, отображающий по одной обратной косой черте в каждом месте. Чтобы убедиться в этом, можете проверить результат встроенной функции `len`, которая возвращает количество символов в строке независимо от форматов отображения. Если вы подсчитаете символы в выводе `print(path)`, то заметите, что на каждую обратную косую черту приходится по одному символу, а всего в строке 15 символов.

Кроме путей к каталогам в Windows неформатированные строки также обычно применяются в регулярных выражениях (сопоставление текста с образцом, поддерживаемое модулем `re` из глав 4 и 37). Также имейте в виду, что сценарии Python, как правило, могут использовать *прямые* косые черты в путях к каталогам в Windows и

Unix, поскольку Python старается интерпретировать пути переносимым образом (т.е. 'C:/new/text.dat' работает при открытии файла). Однако неформатированные строки удобны, если пути записываются с применением естественного для Windows символа обратной косой черты.



Несмотря на свою роль, даже неформатированная строка не может заканчиваться одиночной обратной косой чертой, потому что обратная косая черта отменит следующей за ней символ кавычки — вы по-прежнему должны отменять окружающий символ кавычки для его внедрения в строку. То есть `r"..."` не является допустимым строковым литералом — неформатированная строка не может заканчиваться на нечетное количество символов обратной косой черты. Если нужно поместить в конце неформатированной строки одиночную обратную косую черту, тогда можете использовать две обратные косые черты, отрезав вторую (`r'1\nb\tc\\'[:-1]`), прикрепить одну вручную (`r'1\nb\tc' + '\\'`) или отказаться от синтаксиса неформатированных строк и просто продублировать обратные косые черты в нормальной строке (`'1\\nb\\\\tc\\'`). Все три формы создают одну и ту же восьмисимвольную строку, содержащую три обратные косые черты.

Утроенные кавычки представляют многострочные блочные строки

До сих пор вы видели в действии одинарные кавычки, двойные кавычки, управляющие последовательности и неформатированные строки. В Python также имеется формат строковых литералов с уточненными кавычками, иногда называемый *блочной строкой*, т.е. синтаксическим удобством для написания многострочных текстовых данных. Такая форма начинается с трех кавычек (одинарных или двойных), за которыми следует любое количество строк текста, и заканчивается теми же самыми уточненными кавычками, что и в начале. Одинарные и двойные кавычки, встроенные в текст строки, могут отменяться, но не обязательно — строка не закончится до тех пор, пока Python не встретит три неотмененных кавычки того же вида, который применялся в начале литерала. Вот пример (здесь ... означает приглашение к продолжению набора за пределами IDLE: не вводите его самостоятельно):

```
>>> mantra = """Always look
...     on the bright
...     side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'
```

Строка разнесена на три экранные строки. Как вы узнали в главе 3, в некоторых интерфейсах интерактивное приглашение изменяется на ... при продолжении набора, но IDLE просто переходит на следующую экранную строку; в книге встречаются листинги в обеих формах, так что экстраполируйте сказанное по мере надобности. В любом случае Python собирает весь текст между уточненными кавычками в единственную многострочную строку с внедренными символами новой строки (\n) в местах, где в коде присутствуют разрывы строк. Как видите, вторая строка в литерале имеет ведущие пробелы, но третья — нет; вы получаете в точности то, что набирали. Чтобы посмотреть, каким образом интерпретируется строка с символами новой строки, ее необходимо вывести с помощью `print`:

```
>>> print(mantra)
Always look
    on the bright
side of life.
```

Фактически строки, заключенные в утроенные кавычки, будут сохранять весь находящийся между ними текст, в том числе любые символы справа от кода, которые могли задумываться как *комментарии*. Поэтому не поступайте так — помещайте комментарии до или после текста в уточненных кавычках либо используйте упомянутую ранее автоматическую конкатенацию соседних строк, с явными символами новой строки в случае необходимости и окружающими круглыми скобками, чтобы разрешить распространение на несколько строк (последняя форма рассматривается более подробно при обсуждении синтаксических правил в главах 10 и 12):

```
>>> menu = """spam # Комментарии здесь добавляются к строке!
... eggs # То же самое
...
>>> menu
'spam # Комментарии здесь добавляются к строке!\neggs # То же самое\n'
>>> menu = (
... "spam\n"
... "eggs\n" # Комментарии здесь игнорируются,
... "eggs\n" # но символы новой строки не являются автоматическими
...
>>> menu
'spam\neggs\n'
```

Строки в уточненных кавычках удобны в любое время, когда в программе нужен *многострочный текст*, скажем, для внедрения в файлы исходного кода Python многострочных сообщений об ошибках или кода HTML, XML либо JSON. Вы можете внедрять такие блоки прямо в свои сценарии, помещая их в уточненные кавычки и не прибегая к услугам внешних текстовых файлов или явной конкатенации символов новой строки.

Строки в уточненных кавычках также обычно применяются для *строк документации*, представляющих собой строковые литералы, которые воспринимаются как комментарии, когда они находятся в определенных местах внутри файла (мы подробно рассмотрим их позже в книге). Они не обязаны быть блоками в уточненных кавычках, но обычно являются таковыми, чтобы сделать возможными многострочные комментарии.

Наконец, строки в уточненных кавычках также иногда используются в качестве “ужасно искусного” способа *временного отключения* строк кода во время разработки (хорошо, на самом деле этот способ не настолько ужасен, а представляет собой довольно распространенную практику в наши дни, но суть не в том). Если вы хотите отключить несколько строк кода и снова выполнить сценарий, то просто поместите три кавычки до и после них:

```
X = 1
"""
import os          # Временно отключить этот код
print(os.getcwd())
"""
Y = 2
```

Я назвал такой прием искусственным из-за того, что Python действительно может создать строку из строк кода, отключенных подобным образом, но вероятно это незначительно скажется на производительности. Кроме того, в случае крупных разделов

кода поступать так легче, чем вручную добавлять символы `#` в начало каждой строки и позже удалять их. Сказанное особенно справедливо, если вы применяете текстовый редактор, который не оснащен специальной поддержкой для редактирования кода Python. В Python практичность часто побеждает эстетику.

Строки в действии

После того, как вы создали строку с помощью рассмотренных выше литеральных выражений, вы почти наверняка захотите что-нибудь с ней делать. В этом и последующих двух разделах демонстрируются строковые выражения, методы и форматирование — широко распространенные инструменты обработки текста в языке Python.

Базовые операции

Давайте начнем с взаимодействия с интерпретатором Python, чтобы проиллюстрировать базовые строковые операции, перечисленные ранее в табл. 7.1. Вы можете выполнять конкатенацию строк, используя операцию `+`, и повторять их с применением операции `*`:

```
% python
>>> len('abc')          # Длина: количество элементов
3
>>> 'abc' + 'def'      # Конкатенация: новая строка
'abcdef'
>>> 'Ni!' * 4          # Повторение: подобно "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

Встроенная функция `len` возвращает длину строки (или любого другого объекта, имеющего длину). Формально сложение двух строк посредством операции `+` создает новый строковый объект с объединенным содержимым ее операндов, а повторение с помощью операции `*` подобно добавлению строки к самой себе несколько раз. В обоих случаях Python позволяет создавать строки произвольных размеров. В Python нет нужды предварительно объявлять что-либо, включая размеры структур данных — вы просто создаете строковые объекты по мере необходимости и даете Python возможность автоматически управлять лежащим в основе пространством памяти (в главе 6 рассказывалось о “сборщике мусора” системы управления памятью Python).

Повторение поначалу может выглядеть слегка непонятным, но оно оказывается полезным в удивительно большом количестве контекстов. Например, для вывода 80 символов минуса вы можете сами посчитать до 80 или поручить это Python:

```
>>> print('----- ... и еще... ---')    # 80 символов -, трудный способ
>>> print('-' * 80)                      # 80 символов -, легкий способ
```

Обратите внимание, что здесь уже работает перегрузка операций: мы используем те же самые операции `+` и `*`, которые выполняют сложение и умножение, когда применяются к числам. Язык Python делает правильную операцию, поскольку ему известны типы объектов в операндах. Но будьте осторожны: правила не настолько либеральны, как вы могли бы ожидать. Скажем, Python не разрешает смешивать числа и строки в выражениях с операцией `+`: `'abc' + 9` приводит к ошибке вместо автоматического преобразования 9 в строку.

Как показано ближе к концу в табл. 7.1, вы также можете проходить по строкам в циклах, используя операторы `for`, которые повторяют действия, и проверять членство для символов и подстрок посредством операции выражения `in`, которая по

существу является поиском. В случае подстрок операция `in` очень похожа на метод `str.find()`, раскрываемый далее в главе, но возвращает булевский результат, а не позицию подстроки (в следующем коде применяется вызов `print` из Python 3.X, который может оставить курсор в положении с небольшим отступом; в Python 2.X взамен используется `print c`):

```
>>> myjob = "hacker"
>>> for c in myjob: print(c, end=' ')    # Проход по элементам с выводом
                                            # каждого (форма Python 3.X)
...
h a c k e r
>>> "k" in myjob                      # Найден
True
>>> "z" in myjob                      # Не найден
False
>>> 'spam' in 'abcsspamdef'          # Поиск подстроки без возвращения позиции
True
```

В цикле `for` переменной по очереди присваиваются элементы из последовательности (здесь строки) и для каждого элемента выполняется один или больше операторов. В действительности переменная `c` становится курсором, проходящим по символам строки. Мы более подробно обсудим итерационные инструменты вроде тех, что перечислены в табл. 7.1, позже в книге (особенно в главах 14 и 20).

Индексация и нарязание

Поскольку строки определяются как упорядоченные коллекции символов, мы можем получать доступ к их компонентам по позиции. В Python символы из строки извлекаются посредством *индексации* – предоставления числового смещения желаемого компонента в квадратных скобках после строки. Результатом будет односимвольная строка в указанной позиции.

Как и в языке C, смещения в Python начинаются с 0 и заканчиваются величиной на единицу меньше длины строки. Тем не менее, в отличие от C язык Python также позволяет извлекать элементы из последовательностей, таких как строки, с применением *отрицательных смещений*. Формально отрицательное смещение добавляется к длине строки, чтобы вывести положительное смещение. Об отрицательных смещениях можно также думать как об отсчете в обратном направлении с конца. Вот пример:

```
>>> S = 'spam'
>>> S[0], S[-2]                      # Индексация с начала или с конца
('s', 'a')
>>> S[1:3], S[1:], S[:-1]           # Нарезание: извлечение сегмента
('pa', 'ram', 'spa')
```

В первой команде определяется четырехсимвольная строка и присваивается имени `S`. В следующей команде она индексируется двумя путями: `S[0]` извлекает элемент по смещению 0 слева – односимвольную строку `'s'`; `S[-2]` получает элемент по смещению 2 с конца, что эквивалентно смещению $(4 + (-2))$ с начала. Смещения и срезы отображаются на ячейки, как показано на рис. 7.1¹.

¹ Более предрасположенные к математике читатели (и студенты в моих группах) временами обнаруживают здесь небольшую асимметрию: крайний слева элемент находится по смещению 0, но крайний справа – по смещению -1. Увы, но в Python отсутствует такое понятие, как отдельное значение -0.

[начало:конец]
Индексы ссылаются на места, где производятся срезы.



По умолчанию используются начало и конец последовательности.

Рис. 7.1. Смещения и срезы: положительные смещения начинаются с левого края (смещение 0 дает первый элемент), а отрицательные – с правого края (смещение -1 дает последний элемент). Смещение любого вида может применяться для получения позиций в операциях индексации и нарезания

В последней команде предыдущего примера демонстрируется *нарезание*, т.е. обобщенная форма индексации, которая возвращает целый *сегмент*, а не одиночный элемент. Пожалуй, лучше всего думать о нарезании как о типе *разбора* (анализа структуры), особенно когда оно применяется к строкам – нарезание позволяет извлечь целый *сегмент* (подстроку) за один шаг.

Срезы могут использоваться для извлечения столбцов данных, отсечения ведущего и хвостового текста и т.п. На самом деле мы исследуем нарезание в контексте разбора текста позже в главе.

Основы нарезания прямолинейны. Когда вы индексируете объект последовательности вроде строки с указанием пары смещений, разделенных двоеточием, Python возвращает новый объект, который содержит непрерывный сегмент, идентифицируемый парой смещений. Левое смещение считается нижней границей (*включающей*), а правое – верхней границей (*исключающей*). То есть Python извлекает элементы, начиная с нижней границы и заканчивая, но не включая верхнюю границу, и возвращает новый объект, содержащий извлеченные элементы. Если левая и правая границы не указаны, тогда по умолчанию для них принимается 0 и длина нарезаемого объекта соответственно.

Скажем, в предыдущем примере срез `S[1:3]` извлекает элементы по смещениям 1 и 2: он захватывает второй и третий элементы, останавливаясь перед четвертым элементом по смещению 3. Срез `S[1:]` получает *все элементы, следующие за первым* – верхняя граница, которая не указана, по умолчанию принимается равной длине строки. Наконец, срез `S[:-1]` извлекает *все элементы кроме последнего* – нижняя граница по умолчанию устанавливается в 0, а -1 ссылается на последний элемент, не включая его.

На первый взгляд все может показаться запутанным, но индексация и нарезание являются простыми в применении и мощными инструментами, как только вы поймете, что к чему. Не забывайте, что если вы не уверены в результатах среза, то опробуйте его интерактивно. В следующей главе вы увидите, что возможно даже изменять целый сегмент другого объекта за один шаг, присваивая срез (хотя и не в случае неизменяемых объектов, подобных строкам). Ниже в справочных целях приведена краткая сводка.

Индексация (`S[i]`) извлекает компоненты по смещениям:

- первый элемент находится по смещению 0;
- отрицательные смещения означают отсчитывание с конца или справа;
- `S[0]` извлекает первый элемент;
- `S[-2]` извлекает второй элемент с конца (подобно `S[len(S)-2]`).

Нарезание ($S[i:j]$) извлекает непрерывные сегменты последовательности:

- верхняя граница является исключающей;
- если границы не указаны, тогда по умолчанию для них принимаются 0 и длина последовательности;
- $S[1:3]$ извлекает элементы, которые начинаются со смещения 1 и заканчиваются по смещению 3, не включая его;
- $S[1:]$ извлекает элементы, начиная со смещения 1 и до конца (длина последовательности);
- $S[:3]$ извлекает элементы, которые начинаются со смещения 0 и заканчиваются по смещению 3, не включая его;
- $S[: -1]$ извлекает элементы, начиная со смещения 0 и заканчивая последним элементом, но не включая его;
- $S[:]$ извлекает элементы, начиная со смещения 0 и до конца — создает копию верхнего уровня S .

Расширенное нарязание ($S[i:j:k]$) принимает шаг

(или страйд (большой шаг)) k , который по умолчанию равен +1:

- позволяет пропускать элементы и менять порядок на противоположный, как объясняется в следующем разделе.

Во втором с конца пункте списка описан очень распространенный прием: создание полной копии верхнего уровня объекта последовательности — объект с тем же значением, но расположенный в отдельном участке памяти (вы больше узнаете о копиях в главе 9). Прием не особенно полезен для неизменяемых объектов вроде строк, но пригодится для объектов, которые могут изменяться на месте, таких как списки.

В следующей главе вы увидите, что синтаксис, используемый для индексации по смещению (квадратные скобки), применяется также и для индексации словарей по ключу; операции выглядят одинаково, но их интерпретации отличаются.

Расширенное нарязание: третий предел и объекты срезов

В Python 2.3 и последующих версиях для выражений срезов имеется поддержка необязательного третьего индекса, используемого в качестве *шага* (иногда называемого *страйдом (большим шагом)*). Шаг добавляется к индексу каждого извлеченного элемента. Теперь развитая форма среза выглядит как $X[I:J:K]$ и означает “извлечь все элементы в X , начиная со смещения I и заканчивая смещением $J-1$, с шагом K ”. Третий предел, K , по умолчанию устанавливается в +1 и поэтому все элементы в срезе обычно извлекаются слева направо. Однако если вы укажете явное значение, то сможете применять третий предел для пропуска элементов или смены порядка их следования на противоположный.

Например, срез $X[1:10:2]$ будет извлекать *каждый второй элемент* из X в рамках смещений 1–9, т.е. он накопит элементы по смещениям 1, 3, 5, 7 и 9. Как обычно, первый и второй пределы по умолчанию принимаются равными 0 и длине последовательности соответственно, так что $X[::2]$ получает *каждый второй элемент* с начала и до конца последовательности:

```
>>> S = 'abcdefghijklmnopqrstuvwxyz'
>>> S[1:10:2]      # Пропуск элементов
'bdfhj'
>>> S[::2]
'acegikmo'
```

Можно также использовать отрицательный страйд для накапливания элементов в противоположном порядке. Скажем, выражение среза "hello"[::-1] возвращает новую строку "olleh" – первые две границы, как и ранее, по умолчанию устанавливаются в 0 и длину последовательности, а страйд -1 указывает, что срез должен идти справа налево, а не слева направо, что обычно принято. По этой причине результатом оказывается *обращение* последовательности:

```
>>> S = 'hello'  
>>> S[::-1]          # Смена порядка следования элементов на противоположный  
'olleh'
```

При отрицательном страйде смысл первых двух границ по существу изменяется на противоположный. Таким образом, срез S[5:1:-1] извлекает элементы со второго по пятый в обратном порядке (результат содержит элементы по смещениям 5, 4, 3 и 2):

```
>>> S = 'abcdefg'  
>>> S[5:1:-1]        # Смысл границ изменяется  
'fdec'
```

Пропуск и обращение порядка вроде показанного являются распространенными сценариями применения срезов с тремя пределами, но за дополнительными деталями обращайтесь к руководству по стандартной библиотеке Python (или проведите несколько экспериментов в интерактивной подсказке). Мы еще вернемся к срезам с тремя пределами, когда будем рассматривать оператор цикла `for`.

Позже в книге вы также узнаете, что нарезание эквивалентно индексации с помощью *объекта среза*, что важно для разработчиков классов, стремящихся поддерживать обе операции:

```
>>> 'spam'[1:3]           # Синтаксис нарезания  
'pa'  
>>> 'spam'[slice(1, 3)]  # Индексация посредством объектов срезов  
'pa'  
>>> 'spam'[::-1]  
'maps'  
>>> 'spam'[slice(None, None, -1)]  
'maps'
```

Что потребует внимания: срезы

Повсюду в книге я буду включать врезки с общепринятыми сценариями применения (вроде этой), чтобы дать вам представление о том, как некоторые из описываемых языковых средств обычно используются в реальных программах. Поскольку вы не в состоянии оценить смысл реалистичных сценариев применения до тех пор, пока не увидите большую часть картины Python, такие врезки неизбежно будут содержать много ссылок на пока еще не рассмотренные темы. По большому счету вы должны воспринимать их как предварительное ознакомление с тем, как абстрактные языковые концепции могут оказаться полезными при решении распространенных задач программирования.

Например, позже будет показано, что аргументы командной строки при запуске программы Python доступны в атрибуте `argv` встроенного модуля `sys`:

```
# Файл echo.py  
import sys  
print(sys.argv)  
  
% python echo.py -a -b -c  
['echo.py', '-a', '-b', '-c']
```

Обычно вас будет интересовать только инспектирование аргументов, следующих за именем программы. Это приводит к типичному сценарию употребления срезов: единственное выражение среза может использоваться для возвращения всех элементов списка кроме первого. Здесь `sys.argv[1:]` возвращает желаемый список, `['-a', '-b', '-c']`. Затем список можно обработать, не заботясь о том, что в самом начале находилось имя программы.

Срезы также часто применяются для очистки строк, прочитанных из входных файлов. Если известно, что последним символом в строке будет конец строки (`\n`), то вы можете избавиться от него с помощью выражения `line[:-1]`, которое извлекает все символы кроме последнего (по умолчанию нижняя граница принимается равной 0). В обоих случаях срезы выполняют работу логики, которая должна быть явной в языках более низкого уровня.

Тем не менее, вызов метода `line.rstrip` для удаления символов новой строки зачастую предпочтительнее, поскольку он оставляет строку незатронутой, если она не содержит в конце символа новой строки – обычная ситуация с файлами, создаваемыми рядом инструментов редактирования текста. Нарезание работает, когда вы уверены, что строка заканчивается надлежащим образом.

Инструменты преобразования строк

Один из девизов языка Python заключается в том, что он призывает не поддаваться соблазну выдвигать какие-либо предположения. Как первый пример, вы не можете складывать число и строку в Python, даже если эта строка выглядит похожей на число (т.е. содержит только цифры):

```
# Python 3.X
>>> "42" + 1
TypeError: Can't convert 'int' object to str implicitly
Ошибка типа: нельзя преобразовывать объект int в str неявно

# Python 2.X
>>> "42" + 1
TypeError: cannot concatenate 'str' and 'int' objects
Ошибка типа: нельзя выполнять конкатенацию объектов str и int
```

Именно так было задумано: поскольку `+` может означать и сложение, и конкатенацию, выбор преобразования был бы неоднозначным. Взамен Python трактует это как ошибку. Магия в Python обычно опускается, если она усложняет вашу жизнь.

Тогда что же делать, если ваш сценарий получает число в виде текстовой строки из файла или пользовательского интерфейса? Хитрость в том, что перед тем, как строку можно будет трактовать как число или наоборот, вам необходимо задействовать инструменты преобразования. Например:

```
>>> int("42"), str(42)    # Преобразование из строки в строку
(42, '42')
>>> repr(42) # Преобразование в строку в том виде, как она представлена в коде
'42'
```

Функция `int` преобразует строку в число, а функция `str` преобразует число в его строковое представление (по существу в то, как число выглядит, когда выводится посредством `print`). Функция `repr` (и более старое выражение с обратными апострофами, удаленное из Python 3.X) также преобразует объект в его строковое представление, но возвращает объект в виде строки кода, которую можно выполнить, чтобы

воссоздать объект. Для строк результат содержит кавычки, как если бы он выводится с помощью оператора `print`, что отличается по форме между линейками Python:

```
>>> print(str('spam'), repr('spam')) # Python 2.X: print str('spam'), repr('spam')
spam 'spam'
>>> str('spam'), repr('spam') # Неформатированное интерактивное отображение
('spam', "'spam'")
```

Дополнительные сведения по данным темам ищите во врезке “Форматы отображения `str` и `repr`” в главе 5. В целом функции `int` и `str` предназначены для преобразований в числа и в строки.

Хотя смешивать строки и числовые типы в операциях вроде `+` нельзя, при необходимости операнды можно преобразовать вручную перед тем, как использовать операцию:

```
>>> S = "42"
>>> I = 1
>>> S + I
TypeError: Can't convert 'int' object to str implicitly
Ошибка типа: нельзя преобразовывать объект int в str неявно
>>> int(S) + I      # Принудительно применить сложение
43
>>> S + str(I)      # Принудительно применить конкатенацию
'421'
```

Похожие встроенные функции поддерживают преобразования чисел с плавающей точкой в строки и обратно:

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)
>>> text = "1.234E-10"
>>> float(text)      # В версиях, предшествующих Python 2.7 и 3.1,
                     # отображает больше цифр
1.234e-10
```

Позже мы дополнительно исследуем встроенную функцию `eval`, которая выполняет строку, содержащую код выражения Python, и потому может преобразовывать строку в объект любого типа. Функции `int` и `float` производят преобразование только в числа, но такое ограничение означает, что они обычно быстрее (и безопаснее, т.к. не принимают произвольные выражения). Как кратко упоминалось в главе 5, выражение форматирования строки также предоставляет способ преобразования чисел в строки. Мы обсудим форматирование далее в главе.

Преобразования кодов символов

Что касается преобразований, то также возможно преобразовывать одиночный символ в его внутренний целочисленный код (скажем, в байтовое значение ASCII), передавая символ встроенной функции `ord`, которая возвратит действительное двоичное значение, используемое для представления соответствующего символа в памяти.

Функция `chr` выполняет обратное преобразование, принимая целочисленный код и возвращая надлежащий символ:

```
>>> ord('s')
115
>>> chr(115)
's'
```

Формально обе функции преобразуют символ в и из его порядкового числа или “кодовой точки” Unicode, которая представляет собой просто номер в таблице символов. Для текста ASCII результат будет знакомым 7-битным целым, которое умещается в один байт памяти, но диапазон кодовых точек текста Unicode других видов может быть шире (таблицы символов и Unicode более подробно обсуждаются в главе 37). При необходимости вы можете использовать цикл, чтобы применить эти функции ко всем символам в строке. Такие инструменты также можно использовать для выполнения математических действий, основанных на строках. Например, чтобы перейти к следующему символу, его нужно преобразовать и выполнить математическое действие с целым числом:

```
>>> s = '5'  
>>> s = chr(ord(s) + 1)  
>>> s  
'6'  
>>> s = chr(ord(s) + 1)  
>>> s  
'7'
```

По крайней мере для односимвольных строк такой прием является альтернативой применению встроенной функции `int` для преобразования из строки в целое число (хотя он имеет смысл только в таблицах символов, которые упорядочивают элементы, как того ожидает ваш код!):

```
>>> int('5')  
5  
>>> ord('5') - ord('0')  
5
```

Такие преобразования могут использоваться в сочетании с операторами циклов, которые были представлены в главе 4 и будут подробно раскрыты в следующей части книги, для преобразования строк двоичных цифр в их целочисленные значения. На каждом проходе цикла необходимо умножать текущее значение на 2 и добавлять целочисленное значение следующей цифры:

```
>>> B = '1101'      # Преобразование двоичных цифр в целое число с помощью ord  
>>> I = 0  
>>> while B != '':  
...     I = I * 2 + (ord(B[0]) - ord('0'))  
...     B = B[1:]  
...  
>>> I  
13
```

Операция сдвига влево (`I << 1`) обеспечила бы тот же эффект, что и умножение на 2. Однако мы оставляем это изменение в качестве упражнения из-за того, что циклы пока еще подробно не рассматривались, а также потому, что встроенные функции `int` и `bin`, описанные в главе 5, поддерживают задачи двоичного преобразования, начиная с версий Python 2.6 и 3.0:

```
>>> int('1101', 2)          # Преобразование строки двоичных цифр в целое:  
                           # встроенная функция  
13  
>>> bin(13)              # Преобразование целого в строку двоичных цифр:  
                           # встроенная функция  
'0b1101'
```

При наличии достаточного времени Python стремится автоматизировать большинство распространенных задач!

Изменение строк, часть I

Помните термин “неизменяемая последовательность”? Как вы видели, часть *неизменяемая* означает, что вы не можете изменять строку на месте – скажем, присваивая по индексу:

```
>>> S = 'spam'  
>>> S[0] = 'x'      # Возникает ошибка!  
TypeError: 'str' object does not support item assignment  
Ошибка типа: объект str не поддерживает присваивание в отношении элементов
```

Тогда каким же образом модифицировать текстовую информацию в Python? Чтобы изменить строку, обычно необходимо создать и присвоить новую строку с применением таких инструментов, как конкатенация и нарезание, и затем при желании присвоить результат первоначальному имени строки:

```
>>> S = S + 'SPAM!'    # Чтобы изменить строку, нужно создать новую  
>>> S  
'spamSPAM!'  
>>> S = S[:4] + 'Burger' + S[-1]  
>>> S  
'spamBurger!'
```

В первом примере подстрока добавляется в конец S посредством конкатенации. На самом деле здесь создается новая строка и присваивается переменной S, но вы можете думать об этом как об “изменении” исходной строки. Во втором примере четыре символа заменяются шестью с помощью нарезания, индексации и конкатенации. Как будет показано в следующем разделе, аналогичных результатов можно добиться, используя вызовы методов вроде `replace`:

```
>>> S = 'splot'  
>>> S = S.replace('pl', 'ramal')  
>>> S  
'spamalot'
```

Подобно любой операции, которая выдает новое строковое значение, строковые методы генерируют новые строковые объекты. Если вы хотите сохранить такие объекты, то можете присвоить их переменным. Генерация нового строкового объекта для каждого изменения строки не настолько неэффективна, как может казаться. Вспомните из предыдущей главы, что в ходе выполнения Python автоматически производит сборку мусора (освобождает пространство памяти) в отношении старых неиспользуемых строковых объектов, поэтому новые объекты повторно занимают пространство, где хранились предшествующие значения. Как правило, Python более эффективен, чем вы могли ожидать. Наконец, строить новые текстовые значения можно также с помощью выражений форматирования строк. Следующие два выражения заменяют объекты внутри строки, в некотором смысле преобразуя объекты в строки и изменения исходную строку согласно спецификации формата:

```
>>> 'That is %d %s bird!' % (1, 'dead')      # Выражение форматирования:  
                                                #   все версии Python  
That is 1 dead bird!  
>>> 'That is {0} {1} bird!'.format(1, 'dead')  # Метод форматирования  
                                                #   в Python 2.6, 2.7, 3.X  
'That is 1 dead bird!'
```

Тем не менее, вопреки метафоре подстановки результатом форматирования будет новый строковый объект, а не модифицированный старый. Мы обсудим форматирова-

ние позже в главе; как выяснится, форматирование оказывается более универсальным и удобным, нежели вытекает из приведенного примера. Однако поскольку второй из предшествующих вызовов предлагается в виде метода, прежде чем продолжить исследование форматирования, давайте ознакомимся со строковыми методами.



Как кратко упоминалось в главе 4 и будет подробно раскрыто в главе 37, в версиях Python 3.0 и 2.6 появился новый строковый тип `bytearray`, который является изменяемым и потому допускает модификацию на месте. На самом деле объекты `bytearray` — не строки текста; они представляют собой последовательности небольших 8-битных целых чисел. Тем не менее, они поддерживают большинство тех же операций, что и нормальные строки, и выводятся как символы ASCII при отображении. Соответственно они предоставляют еще один вариант для хранения крупных объемов простого 8-битного текста, который требует частых изменений (более развитые типы текста Unicode подразумевают применение других методик). В главе 37 также будет показано, что функции `ord` и `chr` поддерживают символы Unicode, которые не могут быть сохранены в одиночных байтах.

Строковые методы

В дополнение к операциям выражений строки предоставляют набор *методов*, которые реализуют более сложные задачи. Выражения и встроенные функции в Python могут работать с диапазоном типов, но методы в основном *специфичны для типов объектов* — например, строковые методы работают только со строковыми объектами. В Python 3.X наборы методов некоторых типов пересекаются (скажем, многие типы имеют методы `count` и `copy`), но они по-прежнему более специфичны, чем остальные инструменты.

Синтаксис вызова методов

Как кратко обсуждалось в главе 4, методы — это просто функции, которые ассоциированы с определенными объектами и действуют на них. Формально они являются атрибутами, прикрепленными к объектам, которые ссылаются на вызываемые функции, всегда имеющие в своем распоряжении подразумеваемый объект. Говоря точнее, функции представляют собой пакеты кода, а вызовы методов комбинируют две операции — извлечение атрибута и вызов.

Извлечение атрибутов

Выражение в форме `объект.атрибут` означает “извлечь значение атрибута в объекте”.

Выражения вызовов

Выражение в форме `функция(аргументы)` означает “вызвать код функции, передав ей ноль или большее количество разделенных запятыми объектов аргументов, и возвратить результирующее значение функции”.

Объединение двух операций вместе позволяет вызвать метод объекта. Выражение вызова метода:

`объект.метод(аргументы)`

оценивается слева направо — Python сначала извлечет метод из объекта и затем вызовет его, передавая объект и аргументы. Или простыми словами выражение вызова метода означает следующее:

Вызвать метод для обработки объекта с аргументами.

Если метод вычисляет результат, тогда он также поступает обратно как результат всего выражения вызова метода. Вот более осозаемый пример:

```
>>> S = 'spam'  
>>> result = S.find('pa')      # Вызов метода find для поиска 'pa' в строке S
```

Такое отображение остается справедливым для методов встроенных типов и определяемых пользователем классов, которые мы исследуем позже. В этой части книги вы увидите, что большинство объектов имеют вызываемые методы, и все они доступны с использованием того же самого синтаксиса вызова методов. Как будет показано в последующих разделах, для вызова метода объекта необходим существующий объект; без такого объекта методы не могут быть выполнены (и имеют мало смысла).

Методы строк

В табл. 7.3 приведена сводка по методам и шаблонам вызовов для встроенных строковых объектов в Python 3.3; они часто меняются, поэтому проверяйте руководство по стандартной библиотеке Python на предмет самого актуального списка или запускайте `dir` или `help` на любой строке (либо имени типа `str`) интерактивным образом. Строковые методы Python 2.X слегка варьируются; например, они включают `decode` из-за их отличающейся обработки данных Unicode (то, что мы обсудим в главе 37). В табл. 7.3 имя `S` – это строковый объект, а необязательные аргументы помещены в квадратные скобки.

Таблица 7.3. Вызовы строковых методов в Python 3.3

<code>S.capitalize()</code>	<code>S.ljust(width [, fill])</code>
<code>S.casefold()</code>	<code>S.lower()</code>
<code>S.center(width [, fill])</code>	<code>S.lstrip([chars])</code>
<code>S.count(sub [, start [, end]])</code>	<code>S.maketrans(x[, y[, z]])</code>
<code>S.encode([encoding [,errors]])</code>	<code>S.partition(sep)</code>
<code>S.endswith(suffix [, start [, end]])</code>	<code>S.replace(old, new [, count])</code>
<code>S.expandtabs([tabsize])</code>	<code>S.rfind(sub [,start [,end]])</code>
<code>S.find(sub [, start [, end]])</code>	<code>S.rindex(sub [, start [, end]])</code>
<code>S.format(fmtstr, *args, **kwargs)</code>	<code>S.rjust(width [, fill])</code>
<code>S.index(sub [, start [, end]])</code>	<code>S.rpartition(sep)</code>
<code>S.isalnum()</code>	<code>S.rsplit([sep[, maxsplit]])</code>
<code>S.isalpha()</code>	<code>S.rstrip([chars])</code>
<code>S.isdecimal()</code>	<code>S.split([sep [,maxsplit]])</code>
<code>S.isdigit()</code>	<code>S.splitlines([keepends])</code>
<code>S.isidentifier()</code>	<code>S.startswith(prefix [, start [, end]])</code>
<code>S.islower()</code>	<code>S.strip([chars])</code>
<code>S.isnumeric()</code>	<code>S.swapcase()</code>
<code>S.isprintable()</code>	<code>S.title()</code>
<code>S.isspace()</code>	<code>S.translate(map)</code>
<code>S.istitle()</code>	<code>S.upper()</code>
<code>S.isupper()</code>	<code>S.zfill(width)</code>
<code>S.join(iterable)</code>	

Строковые методы в табл. 7.3 реализуют операции более высокого уровня, такие как разбиение и объединение, преобразования регистра символов, проверка содержимого, поиск и замена подстрок.

Как видите, строковых методов довольно много и мы не имеем возможности раскрыть их все; исчерпывающие сведения ищите в руководстве по библиотеке Python или в справочниках. Однако чтобы помочь вам начать, мы проработаем код, который демонстрирует ряд наиболее распространенных методов в действии, и попутно проиллюстрируем основы обработки текста.

Примеры строковых методов: изменение строк, часть II

Вы уже видели, что поскольку строки неизменяемы, модифицировать их на месте нельзя. Тип `bytearray` поддерживает изменения текста на месте в Python 2.6, 3.0 и последующих версиях, но только для простых 8-битных типов. Мы исследовали способы внесения изменений в строки текста ранее, а здесь снова взглянем на них в контексте строковых методов.

В общем случае для создания нового значения из существующей строки вы конструируете новую строку с помощью операций вроде нарезания и конкатенации.

Скажем, для замены двух символов в середине строки можно применять такой код:

```
>>> S = 'spammy'  
>>> S = S[:3] + 'xx' + S[5:]          # Нарезать сегменты из S  
>>> S  
'spaxxy'
```

Но если нужно только заменить подстроку, тогда можно использовать метод `replace`:

```
>>> S = 'spammy'  
>>> S.replace('mm', 'xx')           # Заменить все mm на xx в S  
>>> S  
'spaxxy'
```

Метод `replace` является более универсальным, чем вытекает из приведенного кода. Он принимает в качестве аргументов исходную подстроку (любой длины) и строку (любой длины), которой должна быть заменена исходная подстрока, и выполняет глобальный поиск и замену:

```
>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')  
'aaSPAMbbSPAMccSPAMdd'
```

В такой роли метод `replace` может применяться как инструмент для реализации замен по образцу (например, в форматированных бланках). Обратите внимание, что теперь мы просто выводим результат, а не присваиваем его какому-то имени — присваивать результаты именам необходимо лишь в случае, когда их желательно сохранить для использования в будущем.

Если требуется заменить одну подстроку фиксированного размера, которая может располагаться по любому смещению, то можно сделать замену снова или выполнить поиск подстроки посредством метода `find` и затем произвести срез:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'  
>>> where = S.find('SPAM')          # Поиск позиции  
>>> where                         # Нашлась по смещению 4  
4
```

```
>>> S = S[:where] + 'EGGS' + S[where+4:]  
>>> S  
'xxxxEGGSxxxxSPAMxxxx'
```

Метод `find` возвращает смещение, по которому обнаруживается подстрока (по умолчанию выполняя поиск с начала), или `-1`, если подстрока не найдена. Как упоминалось ранее, `find` представляет собой операцию поиска подстроки, подобную выражению `in`, но возвращает позицию найденной подстроки.

Еще один вариант предусматривает применение метода `replace` с третьим аргументом, чтобы ограничить его единственной подстановкой:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'  
>>> S.replace('SPAM', 'EGGS')      # Заменить все  
'xxxxEGGSxxxxEGGSxxxx'  
>>> S.replace('SPAM', 'EGGS', 1)    # Заменить одну  
'xxxxEGGSxxxxSPAMxxxx'
```

Обратите внимание, что `replace` каждый раз возвращает новый строковый объект. Поскольку строки неизменяемы, метод никогда по-настоящему не модифицирует исходную строку на месте, хотя и называется `replace`!

Тот факт, что операции конкатенации и метод `replace` каждый раз генерируют новые строковые объекты, на самом деле является потенциальным недостатком их использования для изменения строк. Если нужно применить много изменений к очень крупной строке, то производительность сценария можно улучшить, преобразовав строку в объект, который поддерживает изменения на месте:

```
>>> S = 'spammy'  
>>> L = list(S)  
>>> L  
['s', 'p', 'a', 'm', 'm', 'y']
```

Встроенная функция `list` (вызов конструктора объекта) строит новый список из элементов любой последовательности — в данном случае извлекая символы строки и помещая их в список. Когда строка приобретает такую форму, в нее можно вносить множество изменений без генерации новой копии при каждом изменении:

```
>>> L[3] = 'x'                      # Работает для списков, но не для строк  
>>> L[4] = 'x'  
>>> L  
['s', 'p', 'a', 'x', 'x', 'y']
```

Если после внесения изменений список необходимо преобразовать обратно в строку (скажем, для записи в файл), тогда понадобится вызвать метод `join`:

```
>>> S = ''.join(L)  
>>> S  
'spaxxy'
```

На первый взгляд метод `join` может показаться слегка непонятным. Поскольку `join` представляет собой метод строк (не списков), он вызывается через заданный разделитель. Метод `join` объединяет вместе строки из списка (или другого итерируемого объекта) с разделителями между элементами списка; здесь в качестве разделителя используется пустая строка, чтобы преобразовать список в строку. В более общем случае можно указывать любой строковый разделитель итерируемый объект со строками:

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])  
'eggsSPAMsausageSPAMhamSPAMtoast'
```

На самом деле объединение всех подстрок сразу часто может выполняться быстрее, чем их конкатенация по очереди. Также помните об упомянутой ранее изменяющей строке `bytearray`, которая стала доступной в версиях Python 3.0/2.6 и подробно описывается в главе 37. Из-за того, что ее можно изменять на месте, она предлагает альтернативный способ такой комбинации `list/join` для определенных видов 8-битного текста, который должен часто изменяться.

Примеры строковых методов: разбор текста

Еще одной распространенной ролью строковых методов является простая форма *разбора* текста, т.е. анализа структуры и извлечения подстрок. Для извлечения подстрок по фиксированным смещениям мы можем задействовать методики нарезания:

```
>>> line = 'aaa bbb ccc'  
>>> col1 = line[0:3]  
>>> col3 = line[8:]  
>>> col1  
'aaa'  
>>> col3  
'ccc'
```

Поля данных здесь расположены по фиксированным смещениям и потому могут быть вырезаны из исходной строки. Такой прием подходит для разбора при условии, что компоненты данных имеют фиксированные позиции. Если взамен данные разделены посредством какого-нибудь разделителя, тогда извлечь компоненты можно путем разбиения. Такой прием будет работать, даже когда данные находятся в произвольных позициях внутри строки:

```
>>> line = 'aaa bbb ccc'  
>>> cols = line.split()  
>>> cols  
['aaa', 'bbb', 'ccc']
```

Строковый метод `split` разбивает строку на список подстрок согласно строке разделителя. В приведенном примере разделитель не указан, поэтому по умолчанию принимаются пробельные символы – строка разбивается по группам из одного или большего числа пробелов, табуляций и новых строк, а результатом будет список подстрок. В других приложениях данные могут отделяться с помощью более ясных разделителей. В следующем примере строка разбивается (и тем самым разбирается) по запятым, которые представляют собой частый разделитель в данных, возвращаемых рядом инструментов для работы с базами данных:

```
>>> line = 'bob,hacker,40'  
>>> line.split(',')  
['bob', 'hacker', '40']
```

Разделители могут быть длиннее одного символа:

```
>>> line = "i'mSPAMaSPAMlumberjack"  
>>> line.split("SPAM")  
["i'm", 'a', 'lumberjack']
```

Несмотря на то что возможности разбора при нарезании и разбиении ограничены, обе операции выполняются очень быстро и способны решать рутинные задачи, связанные с извлечением текста. Текстовые данные с разделителями-запятыми являются частью файлового формата CSV; более развитые инструменты в этой области доступны в модуле `csv` из стандартной библиотеки Python.

Другие распространенные строковые методы в действии

Другие строковые методы играют более специфичные роли – например, удаляют пробельные символы в конце строки текста, выполняют преобразование регистра символов, проверяют содержимое и выясняют наличие подстроки в конце или в начале:

```
>>> line = "The knights who say Ni!\n"
>>> line.rstrip()
'The knights who say Ni!'
>>> line.upper()
'THE KNIGHTS WHO SAY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\\n')
True
>>> line.startswith('The')
True
```

Для достижения тех же результатов, которые получаются посредством строковых методов, иногда применяются альтернативные приемы; скажем, операцию членства `in` можно использовать для проверки наличия подстроки, а операции определения длины и нарезания – для имитации `endswith`:

```
>>> line
'The knights who say Ni!\n'
>>> line.find('Ni') != -1      # Поиск через вызов метода или выражение
True
>>> 'Ni' in line
True
>>> sub = 'Ni!\n'
>>> line.endswith(sub)          # Проверка наличия подстроки в конце
                                #   через вызов метода или срез
True
>>> line[-len(sub):] == sub
True
```

Позже в главе будет описан метод форматирования строк `format`, который предоставляет более развитые инструменты подстановки, объединяющие много операций в единственном шаге.

Опять-таки, поскольку для строк доступно так много методов, мы не будем все их рассматривать здесь. Позже в книге вы еще увидите примеры работы со строками, но более подробные сведения ищите в руководстве по библиотеке Python и в других источниках документации или просто самостоятельно экспериментируйте в интерактивном сеансе. Дополнительные подсказки можно получить из результатов вызова `help(S.метод)` для любого строкового объекта `S`; как известно из главы 4, выполнение `help` на `str.метод` вероятно предоставит те же детали.

Обратите внимание, что ни один из строковых методов не принимает *образцы*; для обработки текста, основанной на образцах, должен применяться модуль `re` из стандартной библиотеки Python – расширенный инструмент, который был представлен в главе 4, но по большей части выходит за рамки тематики настоящей книги (еще один краткий пример приводится в конце главы 37). Тем не менее, из-за такого ограничения строковые методы иногда могут выполняться быстрее инструментов, предлагаемых модулем `re`.

Функции первоначального модуля `string` (изъяты из Python 3.X)

История становления строковых методов слегка запутана. В течение примерно первого десятилетия своего существования Python предоставлял стандартный библиотечный модуль по имени `string`, содержащий функции, которые в значительной степени отображаются на текущий набор методов строковых объектов. В ответ на популярное требование в версии Python 2.0 эти функции были сделаны доступными как методы строковых объектов. Однако поскольку слишком много людей написали изрядный объем кода, полагающегося на первоначальный модуль `string`, он был сохранен ради обратной совместимости.

В наши дни вы должны использовать *только строковые методы*, а не первоначальный модуль `string`. Фактически функции первоначального модуля, сформировавшие сегодняшние строковые методы, были полностью удалены из Python 3.X, и вы не должны их применять в новом коде, написанном на Python 2.X или Python 3.X. Тем не менее, из-за того, что вы все еще можете сталкиваться с использованием модуля `string` в старом коде Python 2.X, а книга посвящена Python 2.X и 3.X, здесь приводится краткий обзор.

Развязкой этого наследия стало то, что в Python 2.X по-прежнему формально существуют два способа вызова расширенных операций над строками: вызов методов объектов и вызов функций модуля `string` с передачей объектов в качестве аргументов. Скажем, для переменной `X`, которой присвоен строковый объект, следующий вызов метода объекта:

```
X.метод(аргументы)
```

обычно эквивалентен вызову той же самой операции через модуль `string` (при условии, что модуль уже импортирован):

```
string.метод(X, аргументы)
```

Ниже показан пример схемы с методами в действии:

```
>>> s = 'a+b+c+'  
>>> x = s.replace('+', 'spam')  
>>> x  
'aspambspamcspam'
```

Для доступа к той же операции через модуль `string` в Python 2.X необходимо импортировать модуль (по крайней мере, один раз в процессе) и передать объект:

```
>>> import string  
>>> y = string.replace(s, '+', 'spam')  
>>> y  
'aspambspamcspam'
```

Поскольку подход с модулем так долго был стандартом, а строки – настолько центральный компонент в большинстве программ, вы можете видеть оба шаблона вызовов в коде Python 2.X, который доведется встречать.

Однако, как уже упоминалось, в наши дни вы должны всегда применять вызовы строковых методов, а не более старых функций модуля `string`. На то имеются веские причины помимо того факта, что функции модуля были изъяты из Python 3.X. Во-первых, схема с функциями модуля требует импортирования модуля `string` (для методов импортование не нужно). Во-вторых, вызовы функций модуля длиннее в наборе (когда вы загружаете модуль с помощью `import`, т.е. не используя `from`). Наконец, в-третьих,

функции модуля выполняются медленнее методов (модуль отображает большинство вызовов на методы и тем самым приводит к излишнему вызову в ходе работы).

Сам по себе первоначальный модуль `string` без своих эквивалентов в форме строковых методов предохранен в Python 3.X, потому что он содержит дополнительные инструменты. В их состав входят заранее определенные строковые константы (например, `string.digits`) и система объектов `Template` – не вполне ясный инструмент форматирования, который предшествовал строковому методу `format` и практически здесь не рассматривается. (За деталями обращайтесь к краткому примечанию с его сравнением с другими инструментами форматирования далее в главе, а также к руководству по библиотеке Python.) Тем не менее, если вы действительно хотите модифицировать свой код Python 2.X для применения Python 3.X, тогда должны воспринимать любые вызовы базовых операций над строками просто как призраки из прошлого языка Python.

Выражения форматирования строк

Хотя вы многое можете сделать посредством уже описанных строковых методов и операций над последовательностями, Python также предлагает более развитый способ для объединения задач строковой обработки – *форматирование строк* позволяет выполнять множество подстановок, специфических для типов, в строке за единственный шаг. Оно не является строго обязательным, но может быть удобным, особенно когда форматируемый текст подлежит отображению для пользователей программы. Благодаря изобилию новых идей в мире Python форматирование строк на сегодняшний день доступно в двух вариантах (не считая менее часто употребляемой системы `Template` из модуля `string`, упоминавшейся в предыдущем разделе).

Выражение форматирования строк: '...%s...' % (значения)

Первоначальный прием, доступный с момента появления Python, основан на модели `printf` языка C и повсеместно встречается в большинстве существующего кода.

Вызов метода форматирования строк: '...{}...'.format(значения)

Более новый прием, добавленный в версиях Python 2.6 и 3.0, кое в чем выведен из одноименного инструмента в C#/.NET и частично совпадает по функциональности с выражениями форматирования строк.

Так как вариант с вызовом метода новее, есть шанс, что тот или другой из вариантов может быть объявлен устаревшим и со временем удален. Когда версия Python 3.0 была выпущена в 2008 году, выражение форматирования казалось более вероятным кандидатом для объявления устаревшим в будущих выпусках Python. Действительно, в документации по Python 3.0 предвещали его устаревание в версии Python 3.1 и последующее удаление. С выходом Python 3.3 в 2013 году подобного не случилось, а теперь устаревание и вовсе маловероятно, учитывая широкое использование выражений форматирования – фактически в наши дни они все еще встречаются *тысячами* даже в собственной стандартной библиотеке Python!

Естественно, развитие этой истории зависит от будущей линии поведения пользователей Python. С другой стороны, поскольку в настоящее время законно применять как выражение, так и метод форматирования, и одно из двух может встретиться в коде, с которым придется иметь дело, в книге полностью раскрываются оба приема. Как вы увидите, они оба являются по большому счету *вариациями на одну тему*, хотя

метод обладает рядом дополнительных возможностей (вроде разделителей тысяч), а выражение часто лаконичнее и выглядит второй натурой у большинства программистов на Python.

В самой книге в целях иллюстрации в более поздних примерах используются оба приема. Если у автора есть предпочтение, то он будет в основном держать его в секрете за исключением цитаты из девиза Python, доступного через `import this`: Должен быть один — и предпочтительно только один — очевидный способ делать это.

Если более новый метод форматирования строк не будет значительно лучше первоначального и широко применяемого выражения, то вызванное им *удвоение* требований в базе знаний программистов на Python в данной области представляется неоправданным — и даже не характерным для Python. Программисты не обязаны изучать два сложных инструмента, если они во многом пересекаются. Разумеется, вы должны самостоятельно судить о том, заслуживает ли новый способ форматирования добавления к языку, так что давайте беспристрастно обсудим оба способа.

Основы выражений форматирования

Поскольку выражения форматирования строк появились первыми, мы начнем с них. В Python определена бинарная операция `%` для работы со строками (вы можете вспомнить, что она также вычисляет остаток от деления, или модуль, в случае использования с числами). Когда операция `%` применяется к строкам, она предоставляет простую методику форматирования значений в виде строк согласно определению формата. Вкратце операция `%` предлагает компактный способ записи для множества подстановок строк одновременно вместо создания и конкатенации отдельных частей.

Чтобы сформатировать строку, выполните следующие действия.

- Слева от операции `%` укажите строку формата, содержащую одну или более встроенных целей преобразования, каждая из которых начинается с `%` (например, `%d`).
- Справа от операции `%` укажите объект (или объекты, внедренные в кортеж), который необходимо вставить в строку формата слева на месте цели (или целей) преобразования.

Скажем, в примере форматирования, приведенном ранее в главе, целое число 1 заменяет `%d` в строке формата слева, а строка 'dead' заменяет `%s`. Результатом будет новая строка, отражающая описанные две подстановки, которую можно вывести или сохранить для дальнейшего использования:

```
>>> 'That is %d %s bird!' % (1, 'dead')          # Выражение формата
That is 1 dead bird!
```

Говоря формально, выражения форматирования строк обычно необязательны — в целом вы можете делать похожую работу с помощью множества конкатенаций и преобразований. Однако форматирование позволяет объединить много шагов в единственную операцию. Оно достаточно мощное, чтобы служить оправданием для нескольких дополнительных примеров:

```
>>> exclamation = 'Ni'
>>> 'The knights who say %s!' % exclamation      # Подстановка строки
'The knights who say Ni!'
>>> '%d %s %g you' % (1, 'spam', 4.0)           # Подстановки, специфичные для типов
'1 spam 4 you'
>>> '%s -- %s -- %s' % (42, 3.14159, [1, 2, 3]) # Все типы соответствуют цели %s
'42 -- 3.14159 -- [1, 2, 3]'
```

В первом примере строка 'Ni' подставляется в цель слева, заменяя %s. Во втором примере в целевую строку вставляются три значения. Обратите внимание, что в случае вставки более одного значения их необходимо группировать справа в круглых скобках (т.е. помещать в *кортеж*). Операция % выражения форматирования ожидает в своей правой стороне либо одиночный элемент, либо кортеж из одного или более элементов.

В третьем примере снова вставляются три значения, целое число, число с плавающей точкой и списоковый объект, но все цели слева указаны как %s, что обозначает преобразование в строку. Так как в строку (применимую при выводе) может быть преобразован объект любого типа, с кодом преобразования %s работают объекты всех типов. Таким образом, если только вы не будете задавать какое-то специальное форматирование, то %s часто является единственным кодом, который понадобится запомнить для выражений форматирования.

И снова имейте в виду, что форматирование всегда создает новую строку, а не изменяет строку слева; оно обязано так работать по причине неизменяемости строк. Как и ранее, присвойте результат какой-нибудь переменной, если хотите сохранить его.

Расширенный синтаксис выражений форматирования

Для более сложного форматирования, специфичного для типов, в выражениях форматирования можно использовать любые коды преобразования типов, перечисленные в табл. 7.4; они указываются после символа % в целях подстановки. Программисты на С узнают большинство из них, потому что форматирование строк Python поддерживает все обычные коды формата printf языка С (но возвращает результат вместо его вывода, как делает printf).

Таблица 7.4. Коды типов в форматировании строк

Код	Описание
s	Строка (или строка str(x) любого объекта)
r	То же, что и s, но применяется repr, а не str
c	Символ (int или str)
d	Десятичное число (целое число с основанием 10)
i	Целое число
u	То же, что и d (устарел: больше не является беззнаковым)
o	Восьмеричное целое число (с основанием 8)
x	Шестнадцатеричное целое число (с основанием 16)
X	То же, что и x, но в верхнем регистре
e	Число с плавающей точкой со степенью, в нижнем регистре
E	То же, что и e, но в верхнем регистре
f	Десятичное число с плавающей точкой
F	То же, что и f, но в верхнем регистре
g	Число с плавающей точкой e или f
G	Число с плавающей точкой E или F
%	Литерал % (записывается как %%)

Некоторые коды форматов в табл. 7.4 предоставляют альтернативные способы форматирования того же типа; скажем, `%e`, `%f` и `%g` позволяют форматировать числа с плавающей точкой.

На самом деле цели преобразований в строке формата в левой части выражения поддерживают различные операции преобразования с довольно сложным синтаксисом. Общая структура целей преобразований выглядит следующим образом:

`% [(имя ключа)] [флаги] [ширина] [.точность] код типа`

Символы типов кодов в первой колонке табл. 7.4 находятся в конце такого формата целевой строки.

Между `%` и символом кода типа можно делать любое из описанных ниже действий:

- указывать имя *ключа* для индексации словаря, используемого в правой части выражения;
- перечислять *флаги*, которые указывают аспекты вроде выравнивания влево (`-`), знак числа (`+`), пусто перед положительными числами и `-` для отрицательных чисел (пробел) и дополнение нулями (`0`);
- задавать общую минимальную *ширину* поля для подставляемого текста;
- устанавливать количество цифр, отображаемых после десятичной точки (*точность*), для чисел с плавающей точкой.

Части *ширина* и *точность* также можно записывать как `*` для указания на то, что они должны брать свои значения из следующего элемента во входных значениях правой части выражения (удобно, когда это не известно вплоть до времени выполнения). И если вам не нужны любые дополнительные инструменты, тогда простой код `%s` в строке формата будет заменяться стандартной отображаемой строкой соответствующего значения независимо от его типа.

Более сложные примеры использования выражений форматирования

Синтаксис целей форматирования полностью документирован в стандартных руководствах и справочниках по Python, но для демонстрации распространенных применений давайте рассмотрим несколько примеров. Следующая строка форматирует целые числа по умолчанию и затем шестисимвольное поле с выравниванием влево и дополнением нулями:

```
>>> x = 1234
>>> res = 'integers: ...%d...%-6d...%06d' % (x, x, x)
>>> res
'integers: ...1234...1234 ...001234'
```

Форматы `%e`, `%f` и `%g` отображают числа с плавающей точкой по-разному, как видно во взаимодействии ниже. Здесь `%E` – то же, что и `%e`, но экспонента выводится в верхнем регистре. Кроме того, `g` выбирает форматы по содержимому числа (формально определено использование экспоненциального формата `e`, если показатель степени меньше `-4` или не меньше точности, и в противном случае десятичного формата `f` со стандартной общей точностью 6 цифр):

```
>>> x = 1.23456789
>>> x    # Отображает больше цифр в версиях, предшествующих Python 2.7 и 3.1
1.23456789
```

```
>>> '%e | %f | %g' % (x, x, x)
'1.234568e+00 | 1.234568 | 1.23457'
>>> '%E' % x
'1.234568E+00'
```

Добиться дополнительных эффектов форматирования для чисел с плавающей точкой можно, указывая выравнивание влево, дополнение нулями, знак числа, общую ширину поля и количество цифр после десятичной точки. Для более простых задач можно обойтись простым преобразованием в строки с помощью выражения формата `%s` или показанной ранее встроенной функции `str`:

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23 | 01.23 | +001.2'
>>> '%s' % x, str(x)
('1.23456789', '1.23456789')
```

Когда размеры не известны вплоть до времени выполнения, можно применять вычисляемые ширину и точность, указав для них `*` в строке формата, чтобы значения извлекались из следующего элемента во входных данных справа от операции `%` — в следующем кортеже 4 дает точность:

```
>>> '%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, 0.3333'
```

Если вас заинтересовала такая возможность, то можете самостоятельно поэкспериментировать с некоторыми приведенными примерами и операциями для лучшего их понимания.

Выражения форматирования, основанные на словаре

В качестве более развитого расширения форматирование строк также разрешает целям преобразования в левой части ссылаться на ключи в *словаре*, записанном в правой части, и извлекать соответствующие значения. Это открывает возможность использовать форматирование как инструмент организации шаблона. До сих пор мы лишь кратко упоминали о словарях в главе 4, но вот пример, который демонстрирует основы:

```
>>> '%(qty)d more %(food)s' % {'qty': 1, 'food': 'spam'}
'1 more spam'
```

Здесь `(qty)` и `(food)` слева в строке формата ссылаются на ключи в литерале словаря справа и извлекают связанные с ними значения. Такой прием часто применяется в программах, генерирующих текст HTML или XML. Можно построить словарь значений и подставить их все сразу с помощью единственного выражения форматирования, которое использует ссылки на основе ключей (обратите внимание, что первый комментарий находится выше уточненной кавычки, а потому не добавляется в строку; кроме того, код набирался в IDLE без приглашения ... к продолжению ввода):

```
>>> # Шаблон с целями подстановки
>>> reply = """
Greetings...
Hello %(name)s!
Your age is %(age)s
"""

>>> values = {'name': 'Bob', 'age': 40} # Создание значений для подстановки
>>> print(reply % values)           # Выполнение подстановок
Greetings...
Hello Bob!
Your age is 40
```

Подобный трюк также применяется в сочетании со встроенной функцией `vars`, возвращающей словарь, который содержит все переменные, существующие в месте ее вызова:

```
>>> food = 'spam'  
>>> qty = 10  
>>> vars()  
{'food': 'spam', 'qty': 10, ... плюс встроенные имена, установленные Python... }
```

При использовании в правой части операции `%` это позволяет строке формата ссылаться на переменные по именам – как на ключи словаря:

```
>>> '%(qty)d more %(food)s' % vars()    # Переменные являются ключами в vars()  
'10 more spam'
```

Словари более детально исследуются в главе 8. В главе 5 приводились примеры преобразования строк шестнадцатеричных и восьмеричных цифр посредством кодов `%x` и `%o` выражения форматирования, которые здесь не повторяются. Далее будут демонстрироваться дополнительные примеры выражений форматирования в сравнении с методами форматирования – последней темой в главе, касающейся строк.

Вызовы методов форматирования строк

Как упоминалось ранее, в версиях Python 2.6 и 3.0 появился новый способ форматирования строк, рассматриваемый некоторыми как более специфичный для Python. В отличие от выражений форматирования вызовы методов форматирования не так тесно связаны с моделью `printf` языка С и временами являются более явными в плане намерений. С другой стороны, новая методика по-прежнему опирается на основные концепции `printf` вроде кодов типов и спецификаций форматов. Более того, она в значительной степени пересекается с выражениями форматирования (а иногда требует написания чуть большего объема кода) и во многих практических ролях может быть столь же сложной. Из-за этого на сегодняшний день не существует рекомендаций относительно того, какую из методик лучше применять, и большинству программистов полезно бегло разбираться в обеих схемах. К счастью, они довольно похожи и многие концепции частично совпадают.

Основы методов форматирования

Метод `format` строкового объекта, доступный в Python 2.6, 2.7 и 3.X, базируется на обычном синтаксисе вызова функций, а не на выражении. В частности, он использует в качестве шаблона строку, на которой вызывается, и принимает любое количество аргументов, которые представляют значения, подлежащие подстановке согласно шаблону.

Применение метода `format` требует знания функций и вызовов, но в основном оно прямолинейно. Фигурные скобки внутри строки, на которой метод вызывается, обозначают цели подстановки и аргументы, вставляемые по позиции (например, `{1}`), по ключевому слову (скажем, `{food}`) или по относительной позиции в Python 2.7, 3.1 и более поздних версиях (`{}`). Как вы узнаете во время подробного обсуждения передачи аргументов в главе 18, передавать аргументы функциям и методам можно по позициям или по ключевым словам, а способность Python накапливать произвольное количество позиционных и ключевых аргументов делает возможными такие обобщенные шаблоны вызова методов. Например:

```

>>> template = '{0}, {1} and {2}'                                # По позициям
>>> template.format('spam', 'ham', 'eggs')
'spam, ham and eggs'

>>> template = '{motto}, {pork} and {food}' # По ключевым словам
>>> template.format(motto='spam', pork='ham', food='eggs')
'spam, ham and eggs'

>>> template = '{motto}, {0} and {food}'      # По позициям и ключевым словам
>>> template.format('ham', motto='spam', food='eggs')
'spam, ham and eggs'

>>> template = '{}', {} and {}'                  # По относительным позициям
>>> template.format('spam', 'ham', 'eggs')    # Нововведение Python 3.1 и 2.7
'spam, ham and eggs'

```

Для сравнения выражение форматирования из предыдущего раздела может быть более лаконичным, но в нем используются словари, а не ключевые аргументы, и оно не обеспечивает настолько высокую гибкость в отношении источников значений (что в зависимости от вашей точки зрения может считаться как ценным качеством, так и помехой); мы еще будем сравнивать эти две методики далее в главе:

```

>>> template = '%s, %s and %s'          # То же самое посредством выражения
>>> template % ('spam', 'ham', 'eggs')
'spam, ham and eggs'

>>> template = '%(motto)s, %(pork)s and %(food)s'
>>> template % dict(motto='spam', pork='ham', food='eggs')
'spam, ham and eggs'

```

Обратите внимание на применение здесь `dict()` для создания словаря из ключевых аргументов, представленного в главе 4 и подробно раскрываемого в главе 8; часто он оказывается менее громоздкой альтернативой литералу `{...}`. Естественно, строка, на которой вызывается метод `format`, также может быть литералом, создающим временную строку, а вместо целей могут подставляться объекты произвольных типов, что очень похоже на код `%s` выражения:

```

>>> '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
'3.14, 42 and [1, 2]'

```

Как и в случае с выражением `%` и другими строковыми методами, `format` создает и возвращает новый строковый объект, который можно немедленно вывести либо сохранить для дальнейшей работы (вспомните, что строки неизменяемы, поэтому метод `format` действительно обязан создать новый объект). Форматирование строк предназначено не только для отображения:

```

>>> X = '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
>>> X
'3.14, 42 and [1, 2]'

>>> X.split(' and ')
['3.14', '42', '[1, 2]']

>>> Y = X.replace('and', 'but under no circumstances')
>>> Y
'3.14, 42 but under no circumstances [1, 2]'

```

Добавление ключей, атрибутов и смещений

Подобно выражениям форматирования `%` при расширенном использовании вызовы метода `format` могут стать более сложными. Скажем, строки формата могут со-

держать имена атрибутов объекта и ключей словаря — как в нормальном синтаксисе Python, с помощью квадратных скобок указываются ключи словаря, а посредством точек задаются атрибуты объекта элемента, на который производится ссылка по позиции или по ключевому слову. В первом из следующих примеров словарь индексируется по ключу kind и затем извлекается атрибут platform из объекта импортированного модуля sys. Во втором примере делается то же самое, но вместо позиций применяются ключевые слова:

```
>>> import sys  
>>> 'My {1[kind]} runs {0.platform}'.format(sys, {'kind': 'laptop'})  
'My laptop runs win32'  
>>> 'My {map[kind]} runs {sys.platform}'.format(sys=sys, map={'kind': 'laptop'})  
'My laptop runs win32'
```

В квадратных скобках в строках формата допускается также указывать смещения в списке (или другой последовательности) для выполнения индексации, но внутри строк формата синтаксически работают только положительные смещения, поэтому такая возможность не настолько универсальна, как можно было подумать. Подобно выражениям % для указания отрицательных смещений или срезов либо для использования результатов произвольных выражений в целом выражения придется выполнять за пределами самой строки формата (обратите внимание на применение здесь *parts для распаковки элементов кортежа в индивидуальные аргументы функции, как делалось в главе 5 при изучении дробей; дополнительные сведения ищите в главе 18):

```
>>> somelist = list('SPAM')  
>>> somelist  
['S', 'P', 'A', 'M']  
>>> 'first={0[0]}, third={0[2]}'.format(somelist)  
'first=S, third=A'  
>>> 'first={0}, last={1}'.format(somelist[0], somelist[-1])  
# [-1] не допускается  
'first=S, last=M'  
>>> parts = somelist[0], somelist[-1], somelist[1:3] # [1:3] не допускается  
>>> 'first={0}, last={1}, middle={2}'.format(*parts) # Или '{}' в  
# Python 2.7/3.1+  
"first=S, last=M, middle=['P', 'A']"
```

Расширенный синтаксис методов форматирования

Еще одно сходство с выражениями % связано с тем, что за счет добавления дополнительного синтаксиса в строку формата можно достичь более специфических результатов. Для метода форматирования после обозначения цели с возможно пустой подстановкой мы используем двоеточие, за которым следует спецификатор формата, указывающий размер поля, выравнивание и конкретный код типа. Вот формальная структура того, что может встречаться в качестве цели подстановки в строке формата — все ее четыре части необязательны и должны задаваться без промежуточных пробелов:

```
{имя_поля компонент !флаг_преобразования :спецификатор_формата}
```

Ниже приведено описание синтаксиса цели подстановки.

- *имя_поля* — необязательное число или ключевое слово, идентифицирующее аргумент, которое может быть опущено для применения относительной нумерации аргументов в Python 2.7, 3.1 и более поздних версиях.

- компонент – строка из нуля или большего числа ссылок .имя либо [индекс], используемых для извлечения атрибутов и индексированных значений аргумента, которая может быть опущена для применения всего значения аргумента.
- флаг_преобразования, если присутствует, начинается с символа !, за которым следует r, s или a для вызова на значении встроенных функций repr, str или ascii соответственно.
- спецификатор_формата, если присутствует, начинается с символа :, за которым следует текст, указывающий, каким образом значение должно быть представлено, в том числе такие детали, как ширина поля, выравнивание, дополнение, десятичная точность и т.д., и в конце необязательный код типа данных.

Часть спецификатор_формата после символа двоеточия имеет собственный расширенный формат и формально описывается так (квадратные скобки обозначают необязательные компоненты и записываться буквально не должны):

```
[ [заполнение] выравнивание] [знак] [#][0] [ширина] [,] [.точность] [код_типа]
```

Здесь заполнение может быть любым заполняющим символом кроме { или }; выравнивание – <, >, = или ^ соответственно для выравнивания влево, выравнивания вправо, дополнения символом знака или выравнивания по центру; знак может быть +, - или пробелом; а вариант , (запятая) требует запятой для разделителя тысяч, начиная с версии Python 2.7 и 3.1. Компоненты ширина и точность почти такие же, как в выражении %; кроме того, спецификатор_формата может также содержать вложенные строки формата {} только с именами полей, чтобы получать значения из списка аргументов динамически (что очень похоже на * в выражениях форматирования).

Параметры конструкции код_типа метода практически полностью совпадают с параметрами, которые используются в выражениях % и были ранее перечислены в табл. 7.4, но метод форматирования разрешает также применять код типа b для отображения целых чисел в двоичном виде (эквивалентен использованию встроенной функции bin), код типа % для отображения знака процента и только d для десятичных целых (i либо u не применяются). Обратите внимание, что в отличие от спецификатора %s выражения код типа s здесь требует аргумента в виде строкового объекта; для принятия любого типа обобщенным образом код типа необходимо опустить.

Дополнительные детали о синтаксисе подстановки, которые в главе не затрагивались, ищите в руководстве по библиотеке Python. В дополнение к использованию метода format строки одиночный объект можно форматировать с помощью встроенной функции format(объект, спецификатор_формата), которую метод format применяет внутренне, а также настраивать в определяемых пользователем классах посредством метода перегрузки операции __format__ (см. часть VI).

Более сложные примеры использования методов форматирования

Как вы уже заметили, синтаксис в методах форматирования может оказаться сложным. Поскольку вашим лучшим союзником в таких случаях часто выступает интерактивная подсказка, давайте выполним несколько примеров. В следующих вызовах методов {0:10} означает первый позиционный аргумент в поле шириной 10 символов, {1:<10} – второй позиционный аргумент, выровненный влево в поле шириной 10 символов, а {0.platform:>10} – атрибут platform первого аргумента, выровненный вправо в поле шириной 10 символов (снова обратите внимание на применение dict() для создания словаря из ключевых аргументов, обсуждаемого в главах 4 и 8):

```
>>> '{0:10} = {1:10}'.format('spam', 123.4567)      # В Python 3.3
'spam'      = 123.4567'
>>> '{0:>10} = {1:<10}'.format('spam', 123.4567)
'        spam = 123.4567 '
>>> '{0.platform:>10} = {1[kind]:<10}'.format(sys, dict(kind='laptop'))
'        win32 = laptop '
```

Начиная с версий Python 2.7 и 3.1, во всех случаях номер аргумента можно опускать, если аргументы выбираются слева направо с помощью относительной автоматической нумерации — хотя это делает код менее явным, сводя на нет одно из заявленных преимуществ метода форматирования перед выражением форматирования (о чем пойдет речь в связанной врезке “На заметку!” далее в главе):

```
>>> '{:10} = {:10}'.format('spam', 123.4567)
'spam'      = 123.4567'
>>> '{:>10} = {:<10}'.format('spam', 123.4567)
'        spam = 123.4567 '
>>> '{.platform:>10} = {[kind]:<10}'.format(sys, dict(kind='laptop'))
'        win32 = laptop '
```

Числа с плавающей точкой поддерживают в вызовах методов форматирования те же самые коды типов и особенности форматирования, что и в выражениях %. Скажем, в показанных ниже вызовах {2:g} означает, что третий аргумент форматируется по умолчанию согласно представлению с плавающей точкой g, {1:.2f} указывает формат с плавающей точкой f с только двумя десятичными цифрами после точки, а {2:06.2f} добавляет поле шириной 6 символов и дополнением нулями слева:

```
>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'
>>> '{0:f}, {1:.2f}, {2:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Метод format также поддерживает шестнадцатеричный, восьмеричный и двоичный форматы. На самом деле форматирование строк является альтернативой использованию ряда встроенных функций, которые форматируют целые числа для заданного основания счисления:

```
# Шестнадцатеричный, восьмеричный, двоичный
>>> '{0:X}, {1:o}, {2:b}'.format(255, 255, 255)
'FF, 377, 11111111'

# Другие преобразования в/из двоичного
>>> bin(255), int('11111111', 2), 0b11111111
('0b11111111', 255, 255)

Другие преобразования в/из шестнадцатеричного
>>> hex(255), int('FF', 16), 0xFF #
('0xff', 255, 255)

# Другие преобразования в/из восьмеричного, Python 3.X
>>> oct(255), int('377', 8), 0o377
('0o377', 255, 255)                                # Python 2.X выводит и принимает 0377
```

Параметры форматирования можно либо жестко кодировать в строках формата, либо динамически получать из списка аргументов за счет вложенного синтаксиса формата, очень похожего на синтаксис * в ширине и точности выражений форматирования:

```

>>> '{0:.2f}'.format(1 / 3.0)      # Жестко закодированные параметры
'0.33'
>>> '%.2f' % (1 / 3.0)           # То же самое для выражения
'0.33'
>>> '{0:{1}f}'.format(1 / 3.0, 4) # Получение значений из аргументов
'0.3333'
>>> '%.*f' % (4, 1 / 3.0)       # То же самое для выражения
'0.3333'

```

Наконец, в версиях Python 2.6 и 3.0 появилась новая встроенная функция `format`, которую можно применять для форматирования одиночного элемента. Она является более лаконичной альтернативой использованию строкового метода `format` и напоминает форматирование посредством выражения форматирования `%`:

```

>>> '{0:.2f}'.format(1.2345)      # Строковый метод
'1.23'
>>> format(1.2345, '.2f')        # Встроенная функция
'1.23'
>>> '%.2f' % 1.2345            # Выражение
'1.23'

```

Формально встроенная функция встроенная функция `format` выполняет метод `format` указанного объекта, который метод `str.format` внутренне вызывает для каждого форматируемого элемента. Тем не менее, она все же здесь многословнее первоначального выражения форматирования `%`, что подводит нас к теме следующего раздела.

Сравнение с выражением форматирования `%`

Если вы внимательно изучали предшествующие разделы, то вероятно отметили, что, по крайней мере, для позиционных ссылок и ключей словаря строковый метод `format` выглядит очень похожим на выражение форматирования `%`, в особенности при расширенном применении с кодами типов и дополнительным синтаксисом форматирования. На самом деле в распространенных сценариях использования выражения форматирования может быть *легче* записывать, чем вызовы методов форматирования, особенно когда применяется обобщенная цель вывода строки `%s` и даже с учетом наличия автоматической нумерации полей, добавленной в версиях Python 2.7 и 3.1:

```

# Выражение форматирования: во всех версиях Python 2.X/3.X
print('%s=%s' % ('spam', 42))

# Метод форматирования: в Python 3.0+ и 2.6+
print('{0}={1}'.format('spam', 42))

# С автоматической нумерацией: в Python 3.1+ и 2.7
print('{0}={1}'.format('spam', 42))

```

Как вскоре будет показано, при более замысловатом форматировании подходы имеют тенденцию уравниваться в плане сложности (трудные задачи обычно трудны независимо от подхода), а некоторые считают метод форматирования избыточным, учитывая широкую распространенность выражения.

С другой стороны, метод форматирования также предлагает несколько потенциальных преимуществ. Например, первоначальное выражение `%` неспособно обрабатывать ключевые слова, ссылки на атрибуты и коды двоичных типов, хотя с помощью ссылок на ключи словаря в строках формата `%` часто можно достичь подобных целей.

Чтобы посмотреть, как две методики частично перекрывают друг друга, сравним следующие выражения % с приведенными ранее эквивалентными вызовами метода format:

```
>>> '%s, %s and %s' % (3.14, 42, [1, 2])           # Произвольные типы
'3.14, 42 and [1, 2]'

>>> 'My %(kind)s runs %(platform)s' % {'kind': 'laptop', 'platform': sys.platform}
'My laptop runs win32'

>>> 'My %(kind)s runs %(platform)s' % dict(kind='laptop', platform=sys.platform)
'My laptop runs win32'

>>> somelist = list('SPAM')
>>> parts = somelist[0], somelist[-1], somelist[1:3]
>>> 'first=%s, last=%s, middle=%s' % parts
"first=S, last=M, middle=['P', 'A']"
```

Когда применяется более замысловатое форматирование, эти две методики приближаются к равенству с точки зрения сложности, но если вы сравните показанные ниже выражения с перечисленными ранее эквивалентными вызовами метода format, то снова обнаружите, что выражения % имеют тенденцию быть чуть проще и лаконичнее. Вот случай для Python 3.3:

```
# Добавление специфического форматирования

>>> '%-10s = %10s' % ('spam', 123.4567)
'spam      = 123.4567'

>>> '%10s = %-10s' % ('spam', 123.4567)
'      spam = 123.4567'

>>> '%(plat)10s = %(kind)-10s' % dict(plat=sys.platform, kind='laptop')
'      win32 = laptop'

# Числа с плавающей точкой

>>> '%e, %.3e, %g' % (3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'
>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'

# Шестнадцатеричный и восьмеричный форматы, но не двоичный (см. далее)

>>> '%x, %o' % (255, 255)
'ff, 377'
```

Метод format обладает рядом расширенных возможностей, которые отсутствуют в выражении %, но даже для более запутанного форматирования, похоже, два подхода по существу будут уравниваться в плане сложности. Скажем, ниже демонстрируются одинаковые результаты, сгенерированные обеими методиками, с размерами полей и выравниваниями, а также различными способами ссылки на аргументы:

```
# Жестко закодированные ссылки в обоих случаях

>>> import sys

>>> 'My {1[kind]:<8} runs {0.platform:>8}'.format(sys, {'kind': 'laptop'})
'My laptop    runs    win32'

>>> 'My %(kind)-8s runs %(plat)8s' % dict(kind='laptop', plat=sys.platform)
'My laptop    runs    win32'
```

На практике жесткое кодирование ссылок в программах встречается с меньшей вероятностью, чем выполнение кода, который заблаговременно создает набор данных для подстановки (например, накапливает данные из формы ввода или базы данных с целью подстановки в HTML-шаблон всех за раз). Когда мы учитываем общую практику в примерах, сравнение между методом `format` и выражением `%` становится даже более ясным:

```
# Заблаговременное создание данных двумя способами
>>> data = dict(platform=sys.platform, kind='laptop')
>>> 'My {kind:<8} runs {platform:>8}'.format(**data)
'My laptop    runs    win32'
>>> 'My %(kind)-8s runs %(platform)8s' % data
'My laptop    runs    win32'
```

Как будет показано в главе 18, здесь `**data` в вызове метода является специальным синтаксисом, который распаковывает словарь ключей и значений в индивидуальные присваивания ключевых слов “имя=значение”, так что на них можно ссылаться по имени в строке формата. Это еще одна неизбежная и крайне концептуальная *отережающая ссылка* на инструменты вызова функций, что может быть очередным недостатком метода `format` в целом, особенно для новичков.

Однако, как обычно, сообществу языка Python придется решать, что с течением времени зарекомендовало себя лучше – выражения `%`, вызовы метода `format` или же инструментальный набор с обеими методиками. Самостоятельно поэкспериментируйте с методиками, чтобы получить представление о том, что они предлагают, и обязательно просмотрите руководства по библиотеке для Python 2.6, 3.0 и последующих версий.



Улучшения строкового метода `format` в версиях Python 3.1 и 2.7. В версиях Python 3.1 и 2.7 добавился синтаксис разделителя тысяч для чисел, который позволяет вставлять запятые между группами из трех цифр. Чтобы разделитель тысяч заработал, поместите запятую перед кодом типа, а также между шириной и точностью, если они присутствуют:

```
>>> '{0:d}'.format(999999999999)
'999999999999'
>>> '{0:,d}'.format(999999999999)
'999,999,999,999'
```

Кроме того, в версиях Python 3.1 и 2.7 целям подстановки автоматически назначаются относительные номера, если они не указываются явно, хотя такое расширение неприменимо во всех сценариях использования и может нивелировать одно из главных преимуществ метода форматирования – более ясный код:

```
>>> '{:,d}'.format(999999999999)
'999,999,999,999'
>>> '{:,d} {:,d}'.format(9999999, 8888888)
'9,999,999 8,888,888'
>>> '{:,.2f}'.format(296999.2567)
'296,999.26'
```

Дополнительные сведения ищите в пояснениях к выпуску Python 3.1. Также просмотрите примеры функций вставки запятых и форматирования денежных величин [formats.py](#) в главе 25 для ознакомления с про-

стыми решениями вручную, которые можно импортировать и применять в версиях, предшествующих Python 3.1 и 2.7. В программировании это довольно характерная ситуация — проще самостоятельно реализовать новую функциональность в вызываемой, многократно используемой и настраиваемой функции, а не полагаться на фиксированный набор встроенных инструментов:

```
>>> from formats import commas, money
>>> '%s' % commas(999999999999)
'999,999,999,999'
>>> '%s %s' % (commas(9999999), commas(8888888))
'9,999,999 8,888,888'
>>> '%s' % money(296999.2567)
'$296,999.26'
```

Как обычно, простая функция вроде `commas` может применяться и в более сложных контекстах, таких как инструменты итерации, которые мы встречали в главе 4 и полностью изучим в последующих главах:

```
>>> [commas(x) for x in (9999999, 8888888)]
['9,999,999', '8,888,888']
>>> '%s %s' % tuple(commas(x) for x in (9999999, 8888888))
'9,999,999 8,888,888'
>>> ''.join(commas(x) for x in (9999999, 8888888))
'9,999,9998,888,888'
```

Хорошо это или плохо, но разработчики на Python зачастую предпочитают добавлять встроенные инструменты для специальных случаев, а не использовать обобщенные методики разработки — компромисс, который мы исследуем в следующем разделе.

Для чего используется метод `format`?

Теперь, когда пройден такой путь, для сравнения и противопоставления двух методик форматирования я хочу также объяснить, почему временами у вас может возникать желание применять метод `format`. Короче говоря, невзирая на то, что метод форматирования иногда требует больше кода, он также:

- располагает несколькими дополнительными возможностями, отсутствующими в самом выражении `%` (хотя выражение `%` может использовать альтернативы);
- обладает более гибким синтаксисом ссылки на значения (хотя он может оказаться излишним, а выражение `%` часто имеет эквиваленты);
- может делать ссылки на значения для подстановки более явными (хотя теперь это необязательно);
- обменивает операцию на более значащее имя метода (хотя это и более многословно);
- не допускает разного синтаксиса для одиночных и множественных значений (хотя практика наводит на мысль, что это тривиально);
- как функция может применяться в местах, где выражение не может (хотя односрочная функция разрешает этот спор).

Несмотря на то что на сегодняшний день доступны обе методики и выражение форматирования по-прежнему широко используется, популярность метода `format` со временем может вырасти и в будущем он получит больше внимания со стороны разработчиков на Python. Кроме того, при наличии в языке выражения и метода в коде, с которым вы столкнетесь, может встретиться любой из них, поэтому вам надлежит понимать обе методики. Но поскольку в настоящее время вы все еще можете делать выбор, когда пишете новый код, давайте кратко остановимся на компромиссах, прежде чем закрывать данную тему.

Дополнительные возможности:

“батарейки” для специальных случаев или универсальные методики

Вызов метода поддерживает несколько дополнительных возможностей, которые отсутствуют в выражении, такие как коды двоичных типов и (начиная с Python 2.7/3.1) группирование тысяч. Тем не менее, выражение форматирования обычно позволяет добиться тех же результатов другими способами. Вот сценарий для двоичного форматирования:

```
>>> '{0:b}'.format((2 ** 16) - 1)    # Код двоичного формата в методе (только)
'1111111111111111'
>>> '%b' % ((2 ** 16) - 1)
ValueError: unsupported format character 'b'...
Ошибка значения: неподдерживаемый символ формата b...
>>> bin((2 ** 16) - 1)              # Но работают другие более универсальные варианты
'0b1111111111111111'
>>> '%s' % bin((2 ** 16) - 1)      # Подходит для метода и выражения
'0b1111111111111111'
>>> '{}'.format(bin((2 ** 16) - 1)) # С относительной нумерацией Python 2.7/3.1+
'0b1111111111111111'
>>> '%s' % bin((2 ** 16) - 1)[2:]  # Вырезание 0b для получения точного эквивалента
'1111111111111111'
```

Выше во врезке “На заметку!” было показано, что универсальные функции способны аналогичным образом замещать возможность группирования тысяч метода `format` и более полно поддерживают настройку. В данном случае простая *восьмистрочная* многократно используемая функция приносит ту же пользу без синтаксиса для специальных случаев:

```
# Новая возможность метода str.format в Python 3.1/2.7
>>> '{:,d}'.format(999999999999)
'999,999,999,999'
# Но % позволяет добиться того же с помощью простой функции
>>> '%s' % commas(999999999999)
'999,999,999,999'
```

Дополнительные сведения ищите в упомянутой врезке “На заметку!”. По существу речь идет о сценарии, похожем на применение функции `bin` для двоичного форматирования, но функция `commas` является определяемой пользователем, а не встроенной. Как таковая, эта методика гораздо более *универсальна*, чем готовые инструменты или специальный синтаксис, добавляемый для единственной цели.

Возможно, данный случай также свидетельствует о тенденции Python (и языка написания сценариев в целом) к тому, чтобы больше полагаться на “батарейки в комплекте” для специальных случаев, нежели на универсальные методики разработки. Такой образ мыслей делает код зависимым от этих “батареек” и ему трудно найти объяснение, если только не рассматривать разработку программного обеспечения как занятие конечных

пользователей. Некоторые программисты предпочли бы научиться кодировать алгоритм для вставки запятых, чем получить в свое распоряжение готовый инструмент.

Оставив в стороне философские рассуждения, отметим, что с практической точки зрения совокупным эффектом отмеченной тенденции в этом случае оказывается дополнительный синтаксис, который придется изучить и запомнить. Учитывая имеющиеся альтернативы, не вполне очевидно, что такие дополнительные возможности методов сами по себе достаточно убедительны, чтобы не вызывать сомнений.

Гибкий синтаксис ссылок: дополнительная сложность и частичное совпадение функциональности

Вызов метода `format` также поддерживает ссылки на *ключи* и *атрибуты* напрямую, что некоторые могут посчитать более высокой гибкостью. Но как было показано в предшествующих примерах, сравнивающих форматирование на основе словаря в выражении `%` и ссылки на атрибуты в методе `format`, они обычно слишком похожи, чтобы гарантировать предпочтение на этом основании. Например, оба приема позволяют ссылаться на то же самое значение много раз:

```
>>> '{name} {job} {name}'.format(name='Bob', job='dev')
'Bob dev Bob'
>>> '%(name)s %(job)s %(name)s' % dict(name='Bob', job='dev')
'Bob dev Bob'
```

Однако в обычной практике выражение кажется таким же простым или проще:

```
>>> D = dict(name='Bob', job='dev')
>>> '{0[name]} {0[job]} {0[name]}'.format(D) # Метод, ссылки на ключи
'Bob dev Bob'
>>> '{name} {job} {name}'.format(**D)           # Метод, аргументы из словаря
'Bob dev Bob'
>>> '%(name)s %(job)s %(name)s' % D           # Выражение, ссылки на ключи
'Bob dev Bob'
```

Справедливости ради следует отметить, что метод имеет еще более специализированный синтаксис подстановки, и другие сравнения могут говорить в пользу одной из методик в небольших масштабах. Но с учетом частичного совпадения функциональности и добавочной сложности можно утверждать, что полезность метода представляется либо незначительной, либо признаком поиска сценариев использования. Во всяком случае, увеличившаяся концептуальная нагрузка на программистов, которым теперь необходимо знать *оба* инструмента, не выглядит четко оправданной.

Явная ссылка на значения:

теперь необязательна и вряд ли будет применяться

Один сценарий использования, где метод `format` яснее (по крайней мере, спорно), описывает ситуацию, когда имеется много значений, подлежащих подстановке в строку формата. Скажем, пример класса `lister.py` из главы 31 подставляет шесть элементов в единственную строку и в данном случае метки позиций `{i}` метода `format` выглядят чуть проще для восприятия, чем спецификаторы `%s` выражения `%`:

```
'\n%s<Class %s, address %s:\n%s%s%s>\n' % (...)          # Выражение
'\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format(...)    # Метод
```

С другой стороны, применение в выражениях `%` ключей словаря способно уменьшить эту разницу. Вдобавок такой сценарий кое в чем считается худшим случаем в пла-

не сложности форматирования и на практике встречается нечасто; более типичные сценарии использования не выглядят настолько однозначными. Кроме того, с выходом версий Python 3.1 и 2.7 нумерованные цели подстановки стали необязательными, когда применяются относительные позиции, что потенциально вообще отменяет указанное преимущество:

```
>>> 'The {0} side {1} {2}'.format('bright', 'of', 'life') # Python 3.X, 2.6+
'The bright side of life'
>>> 'The {} side {} {}'.format('bright', 'of', 'life')      # Python 3.1+, 2.7+
'The bright side of life'
>>> 'The %s side %s %s' % ('bright', 'of', 'life')          # Все версии Python
'The bright side of life'
```

Учитывая краткость второго вызова, он вероятно предпочтительнее первого, но, похоже, сводит на нет часть преимущества метода. Сравните, например, эффект форматирования чисел с плавающей точкой — выражение форматирования по-прежнему лаконичнее и выглядит менее запутанным:

```
>>> '{0:f}, {1:.2f}, {2:05.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 03.14'
>>> '{:f}, {:.2f}, {:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Именованный метод и контекстно-нейтральные аргументы: эстетика или практичность

Метод форматирования также заявляет о пользе от замены операции `%` мнемоническим именем метода `format` и от того, что он не проводит различие между одиночным значением и множеством значений подстановки. На первый взгляд замена `%` именем `format` способно сделать метод более простым для восприятия новичками (`format` может быть легче распознавать, чем многочисленные символы `%`), хотя это вероятно зависит от того, кто читает код, и выглядит незначительным.

Некоторые могут счесть вторую характеристику более существенной — в выражении форматирования *одиночное* значение можно задавать само по себе, но *множество* значений должно заключаться в кортеж:

```
>>> '%.2f' % 1.2345           # Одиночное значение
'1.23'
>>> '%.2f %s' % (1.2345, 99) # Кортеж с множеством значений
'1.23 99'
```

Формально выражение форматирования принимает одиночное значение подстановки или кортеж с одним и более элементов. Как следствие, поскольку одиночный элемент может быть задан либо сам по себе, либо внутри кортежа, подлежащий форматированию кортеж должен предоставляться как вложенный кортеж — возможно редкий, но правдоподобный случай:

```
>>> '%s' % 1.23                # Одиночное значение, само по себе
'1.23'
>>> '%s' % (1.23,)
'1.23'
>>> '%s' % ((1.23,),)
# Одиночное значение, являющееся кортежем
'(1.23,)'
```

С другой стороны, метод форматирования усиливает это, принимая в обоих случаях только общие аргументы функции вместо требования кортежа для множества значений или одиночного значения, которое представляет собой кортеж:

```
>>> '{0:.2f}'.format(1.2345)           # Одиночное значение  
'1.23'  
>>> '{0:.2f} {1}'.format(1.2345, 99)    # Множество значений  
'1.23 99'  
>>> '{0}'.format(1.23)                 # Одиночное значение, само по себе  
'1.23'  
>>> '{0}'.format((1.23,))            # Одиночное значение, являющееся кортежем  
'(1.23,)'
```

Следовательно, метод может быть не таким запутывающим для новичков и вызывать меньше ошибок при программировании. Тем не менее, вопрос кажется довольно незначительным — если вы *всегда* заключаете значения в кортеж и игнорируете вариант без кортежа, то выражение по существу совпадает с вызовом метода. Кроме того, ценой за метод будет раздувание размера кода с целью достижения его режима ограниченного использования. Учитывая широкое применение выражений за всю историю Python, вопрос может считаться больше теоретическим, нежели практическим, и не оправдывать перенос существующего кода на новый инструмент, который настолько похож на то, что он пытается поглотить.

Функции или выражения: незначительное удобство

Последнее логическое обоснование добавления метода `format` — что он является функцией, которая может появляться там, где выражение не допускается, — требует знания большего объема информации о функциях, чем известно к текущему моменту, поэтому здесь мы на нем останавливаться не будем. Достаточно сказать, что метод `str.format` и встроенная функция могут передаваться другим функциям, сохраняться в других объектах и т.д. Поступать так с выражением вроде % напрямую нельзя, но это ограниченная точка зрения, поскольку любое выражение легко поместить в односрочный оператор `def` или `lambda` и превратить его в функцию с теми же свойствами (хотя найти причину подобного решения может оказаться сложнее):

```
def myformat(fmt, args): return fmt % args  # См. часть IV  
myformat('%s %s', (88, 99))                  # Вызов вашего объекта функции  
str.format('{0} {1}', 88, 99)                  # Или вызов встроенной функции  
otherfunction(myformat)                      # Ваша функция — тоже объект
```

В конце концов, здесь не может быть выбора либо-либо. Наряду с тем, что выражение форматирования по-прежнему кажется более распространенным в коде Python, на сегодняшний день для использования доступны как выражения, так и методы форматирования, а потому большинство программистов извлечет выгоду на ближайшие годы, освоив обе методики. Объем работы новичков в языке на данном участке может удвоиться, но на этой ярмарке идей, которую мы называем миром программного обеспечения с открытым кодом, всегда найдется место для других².

² См. также врезку “На заметку!” в главе 31 о дефекте метода `str.format` (или регрессе) в версиях Python 3.2 и 3.3, касающегося обобщенных пустых целей подстановки для атрибутов объектов, которые не определяют обработчик `__format__`. Это повлияло на рабочий пример из предыдущего издания книги. Хотя регресс имеет временный характер, во всяком случае, он подчеркивает, что метод `str.format` продолжает развиваться — еще одна причина поставить под сомнение функциональную избыточность, которую он привносит.



На один больше. Говоря формально, имеются *три* (не два) встроенных в Python инструмента форматирования, если мы включим упомянутый ранее неясный инструмент Template модуля `string`. После исследования двух других инструментов давайте посмотрим, как их можно сравнить. Выражение и метод форматирования также разрешено применять в качестве инструментов организации шаблонов, ссылаясь на значения подстановки по именам через ключи словаря или ключевые аргументы:

```
>>> '%(num)i = %(title)s' % dict(num=7, title='Strings')
'7 = Strings'
>>> '{num:d} = {title:s}'.format(num=7, title='Strings')
'7 = Strings'
>>> '{num} = {title}'.format(**dict(num=7, title='Strings'))
'7 = Strings'
```

Система организации шаблонов модуля `string` тоже позволяет ссылаться на значения по именам, снабженным префиксом \$, подобно ключам словаря или ключевым словам, но не поддерживает все полезные свойства двух других методов — ограничение, обусловленное простотой как главной мотивацией создания инструмента `Template`:

```
>>> import string
>>> t = string.Template('$num = $title')
>>> t.substitute({'num': 7, 'title': 'Strings'})
'7 = Strings'
>>> t.substitute(num=7, title='Strings')
'7 = Strings'
>>> t.substitute(dict(num=7, title='Strings'))
'7 = Strings'
```

За дополнительными сведениями обращайтесь в руководства Python. Возможно, вы встретитесь с использованием `Template` (а также ряда сторонних инструментов) в коде Python; к счастью данная методика проста и применяется достаточно редко, чтобы оправдать ее поверхностное рассмотрение здесь. Наилучшим выбором для большинства новичков на сегодняшний день будет изучение и использование %, `str.format` или обоих инструментов.

Общие категории типов

Теперь, когда мы исследовали первый из объектов коллекций Python, строку, давайте закончим главу определением нескольких общих концепций, применимых к большинству типов, которые мы будем рассматривать в дальнейшем. Касаемо встроенных типов выясняется, что операции работают одинаково для всех типов в той же самой категории, поэтому нам понадобится определить большинство таких идей только один раз. До сих пор мы имели дело только с числами и строками, но поскольку они являются типичными представителями двух из трех главных категорий типов в Python, то о ряде других типов вам уже известно намного больше, чем вы можете подумать.

Типы разделяют наборы операций по категориям

Как вы уже знаете, строки — это неизменяемые последовательности: их нельзя модифицировать на месте (часть *неизменяемые*), и они представляют собой позицион-

но упорядоченные коллекции, доступ к которым производится по смещению (часть последовательности). Так случилось, что все последовательности, изучаемые в данной части книги, откликаются на те же самые операции над последовательностями, которые были показаны в главе при работе со строками – конкатенация, индексация, итерация и т.д. Более формально в Python существуют три главные категории типов (и операций), которые обладают такой обобщенной природой.

Числа (целые, с плавающей точкой, десятичные, дроби, остальные)

Поддерживают сложение, умножение и т.д.

Последовательности (строки, списки, кортежи)

Поддерживают индексацию, нарезание, конкатенацию и т.д.

Отображения (словари)

Поддерживают индексацию по ключу и т.д.

Байтовые строки из Python 3.X и строки Unicode из Python 2.X, упомянутые в начале главы, здесь включаются под универсальным обозначением “строки” (см. главу 37). Множества являются в некотором роде категорией по отношению к самим себе (они не отображают ключи на значения и не считаются позиционно упорядоченными последовательностями), а отображения пока еще не затрагивались (они будут обсуждаться в следующей главе). Однако многие другие типы, которые мы встретим, будут похожи на числа и строки. Например, для любых объектов последовательностей X и Y:

- X + Y создает новый объект последовательности с содержимым обоих операндов;
- X * N создает новый объект последовательности, содержащий N копий операнда последовательности X.

Другими словами, такие операции работают одинаково для последовательностей любого вида, включая строки, списки, кортежи и объекты ряда типов, определяемых пользователем. Единственное отличие в том, что получаемый обратно новый объект результата будет иметь тот же тип, как у операндов X и Y – если производится конкатенация списков, то получается новый список, а не строка. Индексация, нарезание и другие операции над последовательностями также работают одинаково на всех последовательностях; типы обрабатываемых объектов сообщают Python о том, задачу какого вида выполнять.

Изменяемые типы можно модифицировать на месте

Классификация по неизменяемости – важное ограничение, о котором следует знать, хотя оно обычно сбивает с толку новичков. Если тип объекта является неизменяемым, тогда его нельзя модифицировать на месте; на попытку сделать это Python отреагирует генерацией ошибки. Взамен потребуется выполнить код для создания нового объекта, содержащего новое значение. Основные типы в Python подразделяются так, как описано ниже.

Неизменяемые типы (числа, строки, кортежи, фиксированные множества)

Типы объектов в категории неизменяемых не поддерживают модификацию на месте, но мы всегда можем выполнять выражения для создания новых объектов и присваивать их результаты переменным по мере надобности.

Изменяемые типы (списки, словари, множества, байтовые массивы)

Наоборот, изменяемые типы всегда могут модифицироваться на месте с помощью операций, которые не создают новые объекты. Несмотря на возможность копирования таких объектов, изменения на месте поддерживают прямую модификацию.

Как правило, неизменяемые типы обеспечивают некоторую степень целостности за счет гарантирования того, что объект не будет изменен другой частью программы. Вы можете вспомнить, почему это так важно, почитав обсуждение разделяемых ссылок на объекты в главе 6. Чтобы посмотреть, каким образом списки, словари и кортежи участвуют в категориях типов, нам необходимо перейти к следующей главе.

Резюме

В главе мы более подробно рассмотрели строковые объекты. Вы узнали о написании строковых литералов и ознакомились со строковыми операциями, включая операции над последовательностями, вызовы строковых методов и форматирование строк посредством выражений и вызовов методов. Попутно вы более глубоко изучили разнообразные концепции, такие как нарязание, синтаксис вызовов методов и блочные строки в устроенных кавычках. Были также определены некоторые основные идеи, общие для множества типов: например, последовательности разделяют целый набор операций.

В следующей главе мы продолжим наше путешествие по типам исследованием наиболее универсальных коллекций объектов в Python – списков и словарей. Как вы обнаружите, многое из того, что уже изучено до сих пор, будет также применимо и к этим типам. Ранее упоминалось о том, что в финальной части книги мы еще возвратимся к модели строк Python, чтобы конкретизировать детали, связанные с текстом Unicode и двоичными данными, которые интересны далеко не всем программистам на Python. Тем не менее, прежде чем двигаться дальше, вот традиционные контрольные вопросы, которые помогут закрепить пройденный материал.

Проверьте свои знания: контрольные вопросы

1. Можно ли использовать строковый метод `find` для поиска в списке?
2. Можно ли применять выражение среза строки для списка?
3. Как бы вы преобразовали символ в его целочисленный код ASCII? Как бы вы выполнили обратное преобразование из целого числа в символ?
4. Как бы вы могли изменить строку в Python?
5. Для заданной строки `S` со значением "`s,ра,т`" назовите два способа извлечения двух символов из середины строки.
6. Сколько символов имеется в строке "`a\nb\x1f\000d`"?
7. Почему вы использовали бы модуль `string` вместо вызова строковых методов?

Проверьте свои знания: ответы

1. Нет, потому что методы всегда специфичны для типа, т.е. работают только с единственным типом данных. Однако выражения вроде `X+Y` и встроенные функции наподобие `len(X)` являются универсальными и способны работать на множестве типов. В этом случае выражение членства `in` дает результат, аналогичный строковому методу `find`, но может применяться для поиска и в строках, и в списках. В Python 3.X была предпринята попытка сгруппировать методы по категориям (скажем, изменяемые типы последовательностей `list` и `bytearray` имеют похожие наборы методов), но методы по-прежнему более специфичны к типам, чем другие наборы операций.
2. Да. В отличие от методов выражения универсальны и применимы ко многим типам. В данном случае выражение среза на самом деле является операцией над последовательностями – оно работает на объектах последовательностей любого типа, включая строки, списки и кортежи. Единственное отличие связано с тем, что при нарезании списка обратно получается новый список.
3. Встроенная функция `ord(S)` преобразует односимвольную строку в целочисленный код символа; `chr(I)` преобразует целочисленный код в строку. Тем не менее, имейте в виду, что эти целые числа являются лишь кодами ASCII для текста, чьи символы извлекаются только из таблицы символов ASCII. В модели Unicode строки текста фактически представляют собой последовательности кодовых точек Unicode, идентифицируемые целыми числами, которые могут выходить за пределы 7-битного диапазона чисел, зарезервированного кодировкой ASCII (дополнительные сведения о строках Unicode ищите в главах 4 и 37).
4. Строки не могут быть модифицированы; они неизменяемы. Однако похожего результата можно добиться путем создания новой строки – за счет конкатенации, нарезания, выполнения выражений форматирования либо использования вызова метода, подобного `replace`, – и затем присваивания результата исходной переменной.
5. Можно получить срез строки с применением `S[2:4]` или разбить строку по запятой и индексировать результат, используя `S.split(',') [1]`. Чтобы удостовериться в этом, опробуйте приемы в интерактивной подсказке.
6. Шесть. Стока "`a\nb\x1f\000d`" содержит символ а, символ новой строки (`\n`), литеральное значение 31 (шестнадцатеричная управляющая последовательность `\x1f`), литеральное значение 0 (восьмеричная управляющая последовательность `\000`) и символ д. Передайте строку встроенной функции `len`, чтобы проверить это, и выведите результаты функции `ord` для каждого ее символа, чтобы увидеть действительные значения кодовых точек (идентифицирующих чисел). Управляющие последовательности были описаны в табл. 7.2.
7. В наши дни вы никогда не должны применять модуль `string` вместо вызова методов строковых объектов – он объявлен устаревшим и в Python 3.X его вызовы полностью устраниены. Единственной веской причиной использования модуля `string` на сегодняшний день являются остальные его инструменты, такие как заранее определенные константы. Вы также можете заметить, что модуль `string` встречается в очень старом, покрывающемся толстым слоем пыли коде Python (и в книгах из туманного прошлого вроде 1990-х годов).

Списки и словари

Теперь, когда числа и строки изучены, в этой главе мы переходим к более детальным исследованиям таких типов объектов, как *списки* и *словари*, которые представляют собой коллекции других объектов и главными рабочими лошадками почти во всех сценариях Python. Как вы увидите, оба типа обладают удивительной гибкостью: они могут изменяться на месте, увеличиваться и уменьшаться по требованию, а также содержать в себе и быть вложенными в объект любого другого вида. С использованием таких типов мы можем строить и обрабатывать в сценариях произвольно насыщенные информационные структуры.

Списки

Следующая остановка в нашем путешествии по встроенным объектам — *списки* Python. Списки являются наиболее гибким типом объектов упорядоченных коллекций Python. В отличие от строк списки способны содержать объекты любого вида: числа, строки и даже другие списки. Также в отличие от строк списки можно модифицировать на месте путем присваивания по смещениям и по срезам, вызова списковых методов, выполнения операторов удаления и т.п. — они представляют собой *изменяемые* объекты.

Списки Python делают работу многих структур данных типа коллекций, которые вам пришлось бы реализовывать вручную в языках более низкого уровня, таких как C. Ниже кратко описаны основные характеристики списков Python.

Они являются упорядоченными коллекциями произвольных объектов

С точки зрения функциональности списки — это всего лишь места для накопления других объектов, так что их можно трактовать как группы. Списки также поддерживают позиционное упорядочение слева направо для содержащихся внутри них элементов (т.е. они представляют собой последовательности).

Они поддерживают доступ по смещению

Как и в случае строк, извлекать объект компонента из списка можно путем индексации списка по смещению объекта. Поскольку элементы в списке упорядочены по своим позициям, можно также выполнять такие операции, как нарезание и конкатенация.

Они имеют переменную длину, разнородны и допускают произвольно глубокое вложение

В отличие от строк списки способны увеличиваться и уменьшаться на месте (их длины могут варьироваться) и содержать объекты любых видов, а не только односимвольные строки (они разнородны). Из-за того, что списки могут содержать другие сложные объекты, они также поддерживают произвольно глубокое вложение; допускается создавать списки, содержащие списки списков и т.д.

Они относятся к категории “изменяемая последовательность”

В терминах категорий типов списки являются изменяемыми (т.е. могут быть модифицированы на месте) и способны реагировать на все операции над последовательностями, применяемые к строкам, такие как индексация, нарезание и конкатенация. На самом деле операции над последовательностями работают со списками так же, как со строками; разница лишь в том, что операции наподобие конкатенации и нарезания в случае использования со списками возвращают новые списки, а не новые строки. Однако поскольку списки изменяемы, они также поддерживают другие операции, не поддерживаемые строками, такие как удаление и присваивание по индексу, которые изменяют списки на месте.

Они представляют собой массивы ссылок на объекты

Формально списки Python содержат ноль и более ссылок на другие объекты. При наличии опыта работы с другими языками списки могут напоминать массивы указателей (адресов). Извлечение элемента из списка Python производится почти так же быстро, как индексация массива С; внутри стандартного интерпретатора списки в действительности представляют собой массивы, а не связные структуры. Тем не менее, как объяснялось в главе 6, всякий раз, когда ссылка используется, Python следует по ней на объект, поэтому программа имеет дело только с объектами. В случае присваивания объекта компоненту структуры данных или переменной Python всегда сохраняет ссылку на тот же самый объект, а не на его копию (если только копия не запрашивается явно).

В качестве обзора и для справочных целей в табл. 8.1 приведен репрезентативный список распространенных операций над списковыми объектами. Он довольно-таки полон для версии Python 3.3. Ознакомиться с завершенным перечнем списковых методов можно в руководстве по стандартной библиотеке Python либо за счет выполнения вызова `help(list)` или `dir(list)` в интерактивной подсказке, которому можно передать реальный список или слово `list` – имя спискового типа данных. Набор методов здесь особенно подвержен к изменениям – фактически в Python 3.3 появились два новых метода.

Таблица 8.1. Распространенные списковые литералы и операции

Операция	Описание
<code>L = []</code>	Пустой список
<code>L = [123, 'abc', 1.23, {}]</code>	Четыре элемента: индексы 0..3
<code>L = ['Bob', 40.0, ['dev', 'mgr']]</code>	Вложенные подсписки
<code>L = list('spam')</code> <code>L = list(range(-4, 4))</code>	Список элементов итерируемого объекта, список последовательных целых чисел
<code>L[i]</code> <code>L[i][j]</code> <code>L[i:j]</code> <code>len(L)</code>	Индекс, индекс индекса, срез, длина
<code>L1 + L2</code> <code>L * 3</code>	Конкатенация, повторение

Операция	Описание
for x in L: print(x) 3 in L	Итерация, членство
L.append(4) L.extend([5, 6, 7]) L.insert(i, X)	Методы: увеличение
L.index(X) L.count(X)	Методы: поиск
L.sort() L.reverse() L.copy() L.clear()	Методы: сортировка, обращение, копирование (Python 3.3+), очистка (Python 3.3+)
L.pop(i) L.remove(X) del L[i] del L[i:j] L[i:j] = []	Методы, операторы: уменьшение
L[i] = 3 L[i:j] = [4, 5, 6]	Присваивание по индексу, присваивание по срезу
L = [x**2 for x in range(5)] list(map(ord, 'spam'))	Списковые включения и отображения (главы 4, 14, 20)

Записанный как литеральное выражение список имеет вид серии объектов (на самом деле выражений, которые возвращают объекты) в квадратных скобках, разделенных запятыми. Например, во второй строке в табл. 8.1 переменной L присваивается четырехэлементный список. Вложенный список записывается как вложенная серия в квадратных скобках (третья строка), а пустой список — просто как пара квадратных скобок, ничего внутри не содержащая (первая строка)¹.

Многие операции в табл. 8.1 должны выглядеть знакомыми, т.к. они представляют собой те же самые операции над последовательностями, которые ранее применялись к строкам — индексация, конкатенация, итерация и т.д. Списки также реагируют на вызовы методов, специфичных для списков (обеспечивающие действия вроде сортировки, обращения, добавления элементов в конец и других), а также операции изменения на месте (удаление элементов, присваивание по индексам и срезам и т.п.). Списки располагают такими инструментами для модификации, потому что они являются изменяемым типом.

Списки в действии

Вероятно, наилучший способ разобраться в списках — посмотреть на них в работе. Далее с помощью простых взаимодействий с интерпретатором мы исследуем операции, перечисленные в табл. 8.1.

¹ На практике в программах обработки списков вы встретите не особенно много списков, записанных подобным образом. Чаще доведется видеть код, который обрабатывает списки, создаваемые динамически (во время выполнения) из пользовательского ввода, содержимого файлов и т.п. Несмотря на важность овладения синтаксисом литералов, многие структуры данных в Python строятся путем запуска программного кода во время выполнения.

Базовые списковые операции

Поскольку списки являются последовательностями, они поддерживают многие операции, относящиеся к строкам. Скажем, списки реагируют на операции + и * почти как строки — упомянутые операции здесь также означают конкатенацию и повторение, но только результатом будет новый список, а не строка:

```
% python
>>> len([1, 2, 3])           # Длина
3
>>> [1, 2, 3] + [4, 5, 6]    # Конкатенация
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4             # Повторение
['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

Хотя операция + работает одинаково для списков и строк, важно знать, что она ожидает *тот же самый* вид последовательности с обеих сторон — иначе при запуске кода возникнет ошибка. Например, выполнить конкатенацию списка и строки не удастся, если сначала не преобразовать список в строку (используя такие инструменты, как str или форматирование %) или строку в список (посредством встроенной функции list):

```
>>> str([1, 2]) + "34"      # То же, что и "[1, 2]" + "34"
'[1, 2]34'
>>> [1, 2] + list("34")     # То же, что и [1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

Итерация по спискам и списковые включения

В более общем смысле списки реагируют на все операции над последовательностями, которые применялись к строкам в предыдущей главе, включая инструменты итерации:

```
>>> 3 in [1, 2, 3]          # Членство
True
>>> for x in [1, 2, 3]:
...     print(x, end=' ')
...
1 2 3
```

Мы обсудим итерацию for и встроенную функцию range, описанные в табл. 8.1, более подробно в главе 13, т.к. они связаны с синтаксисом операторов. Вкратце циклы for проходят по элементам в любой последовательности слева направо, выполняя один и более операторов для каждого элемента; range производит последовательные целые числа.

Операции в последней строке табл. 8.1, списковые включения и вызовы map детально рассматриваются в главах 14 и 20. Однако их базовое действие прямолинейно; как объяснялось в главе 4, списковые включения — это способ построения нового списка за счет применения выражения к каждому элементу в последовательности (на самом деле в любом итерируемом объекте), который тесно связан с циклами for:

```
>>> res = [c * 4 for c in 'SPAM']  # Списковые включения
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

Такое выражение функционально эквивалентно циклу for, который строит список результатов вручную, но как будет показано в последующих главах, списковые включения проще в написании и в наши дни, вероятно, выполняются быстрее:

```
>>> res = []
>>> for c in 'SPAM':           # Эквивалент спискового включения
...     res.append(c * 4)
...
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

Как было кратко указано в главе 4, встроенная функция `map` делает похожую работу, но применяет функцию к элементам в последовательности и собирает все результаты в новый список:

```
>>> list(map(abs, [-1, -2, 0, 1, 2]))    # Отображение функции по всей
                                              # последовательности
[1, 2, 0, 1, 2]
```

Из-за того, что вы еще не полностью готовы к полному изложению итерации, мы пока отложим его, но позже в главе вы еще встретитесь с подобным выражением включения для словарей.

Индексация, нарезание и матрицы

Поскольку списки являются последовательностями, индексация и нарезание работают для списков тем же самым образом, что и для строк. Тем не менее, результатом индексации списка будет объект любого типа, находящийся по указанному смещению, тогда как нарезание списка всегда дает новый список:

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]                                # Смещения начинаются с нуля
'SPAM!'
>>> L[-2]                               # Отрицательные смещения отсчитываются справа
'Spam'
>>> L[1:]                               # Нарезание извлекает сегменты
['Spam', 'SPAM!']
```

Одно замечание: из-за того, что внутрь списков можно вкладывать списки и объекты других типов, временами требуется выстраивать в цепочку операции индексации для углубления в структуру данных. Скажем, один из простейших способов представления матриц (многомерных массивов) в Python предусматривает их организацию в виде списков с вложенными подсписками. Вот базовый двумерный массив 3×3 , основанный на списках:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

С помощью одного индекса получается целая строка (в действительности вложенный подсписок), а посредством двух индексов — элемент внутри строки:

```
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
>>> matrix = [[1, 2, 3],
              [4, 5, 6],
              ...
              [7, 8, 9]]
>>> matrix[1][1]
5
```

В предыдущем взаимодействии обратите внимание на то, что при желании списки могут естественным образом распространяться на несколько строк, т.к. они заключаются между парой квадратных скобок; здесь ... означает приглашение к продолжению ввода в интерактивной подсказке Python.

Дополнительные сведения о матрицах будут приведены позже в главе при обсуждении их представления на основе словарей, которые могут оказаться более эффективными, когда матрицы почти совершенно пусты. Мы также продолжим развивать это направление в главе 20, где запишем добавочный код для работы с матрицами, особенно с помощью списковых включений. Для мощной численной обработки расширение NumPy, упомянутое в главах 4 и 5, предлагает другие способы поддержки матриц.

Изменение списков на месте

Поскольку списки изменяемы, они поддерживают операции, которые изменяют списоковый объект *на месте*. То есть все операции в текущем разделе модифицируют списоковый объект напрямую, переписывая его старое значение, и не требуют создания новой копии, как пришлось бы поступать в случае строк. Из-за того, что Python имеет дело только со ссылками на объекты, такое разграничение между изменением объекта на месте и созданием нового объекта является существенным; как обсуждалось в главе 6, изменение объекта на месте может оказаться влияние более чем на одну ссылку на него.

Присваивания по индексам и срезам

Содержимое списка можно изменять, присваивая значение либо отдельному элементу (по смещению), либо целому сегменту (по срезу):

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'           # Присваивание по индексу
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more'] # Присваивание по срезу: удаление + вставка
>>> L
# Заменяет элементы 0,1
['eat', 'more', 'SPAM!']
```

Присваивания по индексам и по срезам представляют собой изменения на месте – они модифицируют списоковый объект напрямую, а не генерируют для результата новый списоковый объект. *Присваивание по индексу* в Python работает почти как в С и большинстве других языков: Python заменяет одиночную ссылку на объект, находящуюся по заданному смещению, новой ссылкой.

Присваивание по срезу, последняя операция в предыдущем примере, заменяет целый сегмент списка за один шаг. Из-за некоторой сложности для лучшего понимания ее можно воспринимать как комбинацию двух шагов.

1. **Удаление.** Секция, указанная слева от знака =, удаляется.
2. **Вставка.** Новые элементы, содержащиеся в итерируемом объекте справа от знака =, вставляются в список слева в месте, где была удалена старая секция².

Это не то, что происходит в действительности, но оно помогает прояснить, почему количество вставляемых элементов не обязано совпадать с количеством удаляемых элементов.

² Описание требует уточнения, когда значение и срез во время присваивания частично перекрываются: скажем, L[2:5]=L[3:6] работает нормально, потому что вставляемое значение извлекается до того, как слева происходит удаление.

Например, для заданного списка L из двух и более элементов присваивание L[1:2]=[4, 5] заменяет один элемент двумя – Python сначала удаляет одноэлементную секцию в [1:2] (начинающуюся со смещения 1 и заканчивающуюся смещением 2, но включающую его), а затем вставляет значения 4 и 5 там, где раньше был удаленный срез.

Такой прием также объясняет, почему второе присваивание по срезу из числа показанных ниже на самом деле является вставкой – Python заменяет пустой срез в [1:1] двумя элементами, и почему третье присваивание по срезу фактически представляет собой удаление – Python удаляет срез (элемент по смещению 1) и затем ничего не вставляет:

```
>>> L = [1, 2, 3]
>>> L[1:2] = [4, 5]      # Замена/вставка
>>> L
[1, 4, 5, 3]
>>> L[1:1] = [6, 7]     # Вставка (ничего не заменяет)
>>> L
[1, 6, 7, 4, 5, 3]
>>> L[1:2] = []         # Удаление (ничего не вставляет)
>>> L
[1, 7, 4, 5, 3]
```

В действительности присваивание по срезу заменяет целую секцию, или “столбец”, одновременно – даже если столбец или его замена является пустой. Поскольку длина последовательности, которая присваивается, не обязана совпадать с длиной присваиваемого среза, присваивание по срезу может использоваться для замены (путем перезаписывания), увеличения (путем вставки) или уменьшения (путем удаления) обрабатываемого списка. Это мощная операция, но откровенно говоря, на практике вы будете встречать ее относительно редко. Часто существуют более прямолинейные и понятные способы для замены, вставки и удаления (скажем, конкатенация и списковые методы `insert`, `pop` и `remove`), которым программисты на Python обычно отдают предпочтение.

С другой стороны, операцию присваивания по срезу можно применять как разновидность конкатенации в начале списка – согласно описанию методов в следующем разделе расширение списка более уместно в его конце:

```
>>> L = [1]
>>> L[:0] = [2, 3, 4]      # Вставить все на место :0, пустой срез в начале
>>> L
[2, 3, 4, 1]
>>> L[len(L):] = [5, 6, 7] # Вставить все на место len(L):, пустой срез в конце
>>> L
[2, 3, 4, 1, 5, 6, 7]
>>> L.extend([8, 9, 10])   # Вставить все в конце, именованный метод
>>> L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10]
```

Вызовы списковых методов

Подобно строкам списковые объекты Python также поддерживают вызовы методов, специфичных для типов, многие из которых изменяют обрабатываемый список на месте:

```
>>> L = ['eat', 'more', 'SPAM!']
>>> L.append('please')    # Вызов метода дополнения: добавляет элемент в конец
>>> L
['eat', 'more', 'SPAM!', 'please']
```

```
>>> L.sort()          # Сортировка элементов списка ('S' < 'e')
>>> L
['SPAM!', 'eat', 'more', 'please']
```

Методы были представлены в главе 7. Вкратце они являются функциями (на самом деле атрибутами объекта, которые ссылаются на функции), ассоциированными и действующими на определенных объектах. Методы предоставляют инструменты, специфические к типу; например, рассматриваемые здесь списковые методы большей частью доступны только для списков.

Пожалуй, самым ходовым списковым методом следует считать `append`, который просто прикрепляет одиночный элемент (ссылку на объект) в конец списка. В отличие от конкатенации `append` ожидает, что ему будет передан одиночный объект, а не список. Эффект `L.append(X)` аналогичен `L+[X]`, но тогда как первый вариант изменяет `L` на месте, второй создает новый список³. Метод `sort` упорядочивает элементы в списке и заслуживает отдельного раздела.

Дополнительные сведения о сортировке списков

Еще один распространенный метод, `sort`, упорядочивает список на месте; он использует стандартные критерии сравнения Python (здесь строковые сравнения, но он применим к любому типу) и по умолчанию сортирует в возрастающем порядке. Модифицировать поведение сортировки можно путем передачи *ключевых аргументов* – специального синтаксиса “имя=значение” в вызовах функций, который указывает передачу по имени и часто применяется для задания параметров конфигурации.

В методе `sort` аргумент `reverse` позволяет делать сортировку в убывающем порядке вместо возрастающего, а `key` задает функцию с одним аргументом, которая возвращает значение, подлежащее использованию при сортировке – стандартный преобразователь регистра символов `lower` строкового объекта в следующем примере (хотя более новый преобразователь `casefold` мог бы лучше обрабатывать некоторые типы текста Unicode):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()          # Сортировка со смешанным регистром символов
>>> L
['ABD', 'aBe', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)      # Приведение к нижнему регистру
>>> L
['abc', 'ABD', 'aBe']
>>>
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)    # Изменение порядка сортировки
>>> L
['aBe', 'ABD', 'abc']
```

Аргумент `key` метода `sort` также может быть полезен при сортировке списков словарей для выбора ключа сортировки путем индексирования каждого словаря. Словари будут исследоваться позже в главе, а о ключевых аргументах функций вы узнаете в части IV.

³ В отличие от конкатенации + метод `append` не обязан генерировать новые объекты, поэтому обычно он тоже быстрее +. Метод `append` можно также имитировать с помощью искусственных присваиваний по срезам из предыдущего раздела: `L[len(L):]=[X]` похоже на `L.append(X)`, а `L[:0]=[X]` – на дополнение в начало списка. Оба присваивания удаляют пустой срез и вставляют `X`, быстро изменения `L` на месте подобно `append`. Однако оба они возможно сложнее вызова списковых методов. Например, вызов `L.insert(0, X)` способен добавлять элемент также в начало списка и выглядит значительно понятней; вызов `L.insert(len(L), X)` вставит один объект в конец списка, но с таким же успехом можно применять `L.append(X)`, уменьшив объем набираемого кода!



Сравнение и сортировка в Python 3.X. Как было отмечено в главе 5, в Python 2.X сравнение относительных величин по-разному типизированных объектов (например, строки и списка) работает — язык определяет среди разных типов фиксированный порядок, который является детерминированным, пусть и не эстетически привлекательным. Упорядочение основано на именах участвующих типов: скажем, все целые числа меньше всех строк, потому что "int" меньше, чем "str". Сравнения никогда не преобразуют типы автоматически кроме случаев, когда сравниваются объекты числовых типов.

В Python 3.X ситуация изменилась: сравнение величин объектов смешанных типов вызывает исключение, а не следование фиксированному порядку между типами. Поскольку сортировка внутренне применяет сравнения, это означает, что `[1, 2, 'spam'].sort()` успешно выполнится в Python 2.X, но сгенерирует исключение в Python 3.X. Сортировка не работает по определению.

Кроме того, в Python 3.X больше не поддерживается передача методу `sort` произвольной функции сравнения для реализации разных порядков. Обходной путь предполагает использование ключевого аргумента `key=функция для кодирования трансформаций значений во время сортировки` и ключевого аргумента `reverse=True` для изменения порядка сортировки на убывающий. Так выглядели типичные применения функций сравнения в прошлом.

Одно предостережение: помните о том, что `append` и `sort` изменяют ассоциированный списоковый объект на месте, но не возвращают список в качестве результата (формально оба метода возвращают значение `None`). Если вы запишете что-то вроде `L=L.append(X)`, то не получите модифицированное значение `L` (на самом деле вы вообще утратите ссылку на список!). Когда используются такие атрибуты, как `append` и `sort`, объекты изменяются в форме побочного эффекта, поэтому нет причин присваивать повторно.

Отчасти из-за таких ограничений в последних версиях Python сортировка также доступна в виде встроенной функции, которая сортирует любую коллекцию (не только списки) и возвращает в качестве результата новый список (вместо изменений на месте):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L, key=str.lower, reverse=True)      # Встроенная функция сортировки
['aBe', 'ABD', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True) # Предварительно трансформирует
                                                # элементы: отличается!
['abe', 'abd', 'abc']
```

Обратите внимание на последний пример — мы можем выполнить преобразование в нижний регистр с помощью спискового включения перед сортировкой, но результат не содержит значения исходного списка, как было бы в случае аргумента `key`. Преобразование, указанное в аргументе `key`, применяется временно в течение сортировки, не изменяя исходные значения. Позже вы увидите контексты, в которых встроенная функция `sorted` иногда может оказываться более удобной, чем метод `sort`.

Другие распространенные списковые методы

Как и строки, списки располагают дополнительными методами, которые выполняют другие специализированные операции. Например, `reverse` меняет порядок следования элементов в списке на противоположный (обращает список) на месте,

`extend` вставляет множество элементов в конец списка, а `pop` удаляет элемент из конца списка. Существует также встроенная функция `reversed`, которая работает подобно `sorted` и возвращает новый результирующий объект, но в Python 2.X и 3.X должна быть помещена внутрь вызова `list`, т.к. ее результат является итератором, который выпускает результаты по требованию (мы рассмотрим итераторы позже):

```
>>> L = [1, 2]
>>> L.extend([3, 4, 5])      # Добавление множества элементов в конец
                             # (подобно + на месте)
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                  # Удаление и возврат последнего элемента
                             # (по умолчанию: -1)
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()              # Метод обращения списка на месте
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))       # Встроенная функция обращения списка
                             # с результатом (итератор)
[1, 2, 3, 4]
```

Формально метод `extend` всегда проходит по *итерируемому* объекту и добавляет каждый его элемент, тогда как `append` просто добавляет одиночный элемент в том виде, как есть, не проходя по нему – отличие, которое прояснится в главе 14. Пока достаточно знать, что метод `extend` добавляет множество элементов, а метод `append` – один. В некоторых видах программ списоковый метод `pop` часто используется в сочетании с `append` для реализации структуры *стека LIFO* (last-in-first-out – последний на входе, первый на выходе). Конец списка служит верхушкой стека:

```
>>> L = []
>>> L.append(1)                # Затолкнуть в стек
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()                   # Вытолкнуть из стека
2
>>> L
[1]
```

Метод `pop` принимает также необязательное смещение элемента, который необходимо удалить и возвратить (по умолчанию берется последний элемент со смещением `-1`). Другие списковые методы удаляют элемент по значению (`remove`), вставляют элемент по смещению (`insert`), подсчитывают количество вхождений (`count`) и находят смещение элемента (`index` – поиск *индекса* элемента; не путайте его с индексацией!):

```
>>> L = ['spam', 'eggs', 'ham']
>>> L.index('eggs')           # Индекс объекта (поиск)
1
>>> L.insert(1, 'toast')       # Вставка в позицию
>>> L
['spam', 'toast', 'eggs', 'ham']
>>> L.remove('eggs')          # Удаление по значению
>>> L
```

```
['spam', 'toast', 'ham']
>>> L.pop(1)                                # Удаление по позиции
'toast'
>>> L
['spam', 'ham']
>>> L.count('spam')                         # Количество вхождений
1
```

Обратите внимание, что в отличие от других списковых методов методы `count` и `index` не изменяют сам список, а возвращают информацию о его содержимом. За дополнительными сведениями о списковых методах обращайтесь в документацию или поэкспериментируйте с их вызовами в интерактивной подсказке.

Другие распространенные списковые операции

Из-за того, что списки изменяемы, можно применять оператор `del` для удаления элемента или секции на месте:

```
>>> L = ['spam', 'eggs', 'ham', 'toast']
>>> del L[0]                                 # Удаление одного элемента
>>> L
['eggs', 'ham', 'toast']
>>> del L[1:]                               # Удаление целой секции
>>> L
# То же, что и L[1:] = []
['eggs']
```

Как было показано ранее, поскольку присваивание по срезу представляет собой удаление плюс вставку, можно также удалять секцию списка, присваивая срезу пустой список (`L[i:j]=[]`); Python удаляет срез, заданный слева, и затем ничего не вставляет. С другой стороны, присваивание пустого списка по индексу просто сохраняет ссылку на пустой списоковый объект в указанном месте, но не удаляет элемент:

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[]
```

Хотя все только что обсужденные операции являются типичными, могут появиться дополнительные списковые методы и операции, которые здесь не иллюстрировались. Скажем, набор методов со временем может измениться, и фактически это произошло в версии Python 3.3 – ее новый метод `L.copy()` создает копию верхнего уровня списка, почти как `L[:]` и `list(L)`, но симметрично с методом `copy` во множествах и словарях. Чтобы получить исчерпывающий и актуальный список инструментов типа, вы всегда должны заглядывать в руководства по Python, вызывать функции `dir` и `help` (впервые представленные в главе 4) или обращаться к справочникам, упомянутым в предисловии.

Учитывая часто возникающие недоразумения, я хочу еще раз напомнить вам о том, что все исследованные здесь операции изменения на месте работают только с изменяемыми объектами: они неприменимы в отношении строк (или кортежей, рассматриваемых в главе 9), как бы вы ни старались. Изменяемость – неотъемлемое свойство каждого типа объектов.

Словари

Наряду со списками *словари* являются одним из наиболее гибких типов данных в Python. Если вы рассматриваете списки как упорядоченные коллекции объектов, тогда можете считать словари неупорядоченными коллекциями; главное отличие в том, что в словарях элементы сохраняются и извлекаются по *ключу*, а не по позиционному смещению. В то время как списки способны исполнять роли, подобные массивам в других языках, словари замещают собой записи, поисковые таблицы и объединение иного вида, где имена элементов более содержательны, чем их позиции.

Например, словари могут заменять многие алгоритмы поиска и структуры данных, которые пришлось бы реализовывать вручную в языках более низкого уровня — индексация словаря как крайне оптимизированного встроенного типа является очень быстрой операцией. Иногда словари также выполняют работу записей, структур и таблиц символов, используемых в других языках, могут применяться для представления разреженных (по большей части пустых) структур данных и делать многое другое. Ниже кратко описаны основные характеристики словарей Python.

Они поддерживают доступ по ключу, а не по смещению

Словари временами называют *ассоциативными массивами* или *хешами* (особенно пользователи других языков написания сценариев). Они ассоциируют набор значений с ключами, так что извлекать элемент из словаря можно с использованием ключа, под которым он был первоначально сохранен. Для получения компонентов из словаря применяется та же операция индексации, что и в случае списка, но индекс принимает форму ключа, а не относительного смещения.

Они являются неупорядоченными коллекциями произвольных объектов

В отличие от списка элементы, хранящиеся в словаре, не поддерживают какой-то определенный порядок; на самом деле для обеспечения быстрого поиска Python сохраняет их в псевдослучайном порядке. Ключи представляют символические (не физические) позиции элементов в словаре.

Они имеют переменную длину, разнородны и допускают произвольно глубокое вложение

Подобно спискам словари могут увеличиваться и уменьшаться на месте (без создания новой копии), способны содержать объекты любого типа и поддерживают вложение на любую глубину (могут содержать списки, другие словари и т.д.). Каждый *ключ* способен иметь только одно ассоциированное *значение*, но это значение при необходимости может быть *коллекцией* из множества значений, а отдельное значение может храниться под любым количеством ключей.

Они относятся к категории “изменяемое отображение”

Словари разрешено изменять на месте, выполняя присваивания по индексам (они изменяемые), но они не поддерживают операции над последовательностями, которые работают на строках и списках. Поскольку словари — это неупорядоченные коллекции, то операции, которые полагаются на фиксированный позиционный порядок (скажем, конкатенация, нарезание), не имеют смысла. Взамен словари являются единственным встроенным представлением в категории *отображений* внутри основных типов — объектов, которые отражают ключи на значения. Остальные отображения в Python создаются импортируемыми модулями.

Они представляют собой таблицы ссылок на объекты (хеш-таблицы)

Если списки являются массивами ссылок на объекты, поддерживающими доступ по позиции, то словари – неупорядоченными таблицами ссылок на объекты, которые поддерживают доступ по ключу. Внутренне словари реализованы как хеш-таблицы (структуры данных, обеспечивающие очень быстрое извлечение), которые сначала имеют небольшие размеры и увеличиваются по требованию. Кроме того, при поиске ключей Python задействует оптимизированные алгоритмы хеширования, так что извлечение происходит быстро. Подобно спискам словари хранят ссылки на объекты (не копии, если только они не запрошены явно).

В качестве обзора и для справочных целей в табл. 8.2 приведен репрезентативный список распространенных операций над словарями, относительно полный для Python 3.X. Как обычно, ознакомиться с завершенным списком операций можно в руководстве по стандартной библиотеке Python или за счет выполнения вызова `dir(dict)` или `help(dict)` в интерактивной подсказке, где `dict` – имя типа словаря. В форме литературного выражения словарь записывается как серия пар `ключ: значение`, разделенных запятыми, которая помещается в фигурные скобки⁴. Пустой словарь выглядит как пустой набор фигурных скобок. Словари можно вкладывать, просто записывая один словарь в виде значения внутри другого словаря либо внутри списка или кортежа.

Словари в действии

Как видно в табл. 8.2, словари индексируются по ключу, а на элементы вложенных словарей можно ссылаться по цепочке индексов (ключей в квадратных скобках). Когда Python создает словарь, он сохраняет элементы в любом порядке слева направо, который выберет; чтобы извлечь значение, понадобится предоставить ключ, с которым оно ассоциировано, а не его относительную позицию. Давайте вернемся в интерпретатор и опробуем некоторые словарные операции из табл. 8.2.

Таблица 8.2. Распространенные словарные литералы и операции

Операция	Описание
<code>D = {}</code>	Пустой словарь
<code>D = { 'name': 'Bob', 'age': 40}</code>	Двухэлементный словарь
<code>E = { 'cto': { 'name': 'Bob', 'age': 40}}</code>	Вложение
<code>D = dict(name='Bob', age=40)</code>	Альтернативные методики создания:
<code>D = dict([('name', 'Bob'), ('age', 40)])</code>	ключевые слова, пары “ключ/значение”, упакованные пары “ключ/значение”, списки ключей
<code>D = dict(zip(keyslist, valueslist))</code>	
<code>D = dict.fromkeys(['name', 'age'])</code>	

⁴ Как и в случае списков, вам не придется особенно часто видеть словари, записанные целиком с использованием литералов. Дело в том, что программам редко известны все данные до их запуска, и более типичны ситуации, когда данные динамически получаются из пользовательского ввода, файлов и т.д. Однако списки и словари увеличиваются по-разному. В следующем разделе вы узнаете, что словари зачастую создаются присваиванием по новым ключам во время выполнения; такой подход неприменим к спискам, которые обычно увеличиваются с помощью вызовов `append` или `extend`.

Операция	Описание
D['name'] E['cto']['age']	Индексирование по ключу
'age' in D	Членство: проверка присутствия ключа
D.keys() D.values() D.items() D.copy() D.clear() D.update(D2) D.get(key, default?) D.pop(key, default?) D.setdefault(key, default?) D.popitem()	Методы: все ключи, все значения, все кортежи ключ+значение, копирование (верхнего уровня), очистка (удаление всех элементов), объединение по ключам, извлечение по ключу; если отсутствует, тогда возвратить стандартное значение (или None), удаление по ключу; если отсутствует, тогда возвратить стандартное значение (или ошибку) установка по ключу; если отсутствует, тогда ус- тановить в стандартное значение (или None), удаление/возвращение любой пары (ключ, зна- чение); и т.д.
len(D)	Длина: количество сохраненных элементов
D[key] = 42	Добавление ключей, изменение значений, свя- занных с ключами
del D[key]	Удаление элементов по ключу
list(D.keys()) D1.keys() & D2.keys()	Словарные представления (Python 3.X)
D.viewkeys(), D.viewvalues()	Словарные представления (Python 2.7)
D = {x: x*x2 for x in range(10)}	Включения словаря (Python 3.X, 2.7)

Базовые словарные операции

Обычно сначала создается словарь с помощью литерального выражения, а затем в нем сохраняются элементы, доступ к которым впоследствии производится посредством индексации:

```
% python
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}      # Создание словаря
>>> D['spam']                                # Извлечение значения по ключу
2
>>> D                                         # Порядок "перемешан"
{'eggs': 3, 'spam': 2, 'ham': 1}
```

Словарь присваивается переменной D; значение ключа 'spam' – целое число 2 и т.д. Для индексации словарей по ключу применяется тот же самый синтаксис с квадратными скобками, что и при индексации списков по смешению, но здесь он означает доступ по ключу, а не по позиции.

Обратите внимание на окончание примера – во многом подобно множествам *поря-
док слева направо* ключей в словаре будет почти всегда отличаться от порядка, в кото-
ром первоначально вводились элементы. Это сделано намеренно: чтобы реализовать

быстрый поиск ключей (т.е. хеширование), ключи должны быть переупорядочены в памяти. Именно потому операции, предполагающие фиксированный порядок слева направо (например, нарезание, конкатенация), к словарям не применяются; значения можно извлекать только по ключу, но не по позиции. Формально упорядочение является *псевдослучайным* – оно не по-настоящему случайное (вы в состоянии расшифровать его, взяв исходный код Python и потратив массу времени), но оно выбрано произвольно и может варьироваться в зависимости от выпуска и платформы и даже от интерактивного сеанса в версии Python 3.3.

Встроенная функция `len` работает также и со словарями; она возвращает количество элементов, хранящихся в словаре, или, что эквивалентно, длину его списка ключей. Операция членства в словаре `in` позволяет проверять существование ключей, а метод `keys` возвращает все ключи в словаре и может быть удобен при последовательной обработке словарей, но вы не должны полагаться на порядок следования ключей в списке. Тем не менее, поскольку результат `keys` может использоваться как нормальный список, его всегда можно отсортировать, если порядок важен (мы рассмотрим сортировку и словари позже):

```
>>> len(D)                                # Количество элементов в словаре
3
>>> 'ham' in D                            # Проверка членства
True
>>> list(D.keys())                         # Создание нового списка ключей в D
['eggs', 'spam', 'ham']
```

Обратите внимание на второе выражение в листинге. Как упоминалось ранее, проверка членства `in`, применяемая для строк и списков, также работает со словарями – она выясняет, хранится ли ключ в словаре. Формально она работает из-за того, что словари определяют *итераторы* ключей и когда только возможно используют быстрый прямой поиск. Другие типы предлагают итераторы, которые отражают их обычное применение; скажем, файлы имеют итераторы, выполняющие чтение строки за строкой. Мы обсудим итераторы в главах 14 и 20.

Также взгляните на синтаксис последнего выражения в листинге. В Python 3.X нам пришлось поместить `keys` внутрь вызова `list` по сходным причинам – `keys` в Python 3.X возвращает итерируемый объект вместо физического списка. Вызов `list` заставляет его выпустить все значения за раз, так что мы можем вывести их интерактивно, хотя в ряде других контекстов этот вызов не требуется. В Python 2.X метод `keys` создает и возвращает действительный список, поэтому для отображения результата вызов `list` не нужен; позже в главе мы еще вернемся к данной теме.

Изменение словарей на месте

Продолжим наш интерактивный сеанс. Словари, как и списки, являются изменяемыми, так что их можно модифицировать, увеличивать и уменьшать на месте, не создавая новые словари: для изменения или создания элемента необходимо просто выполнить присваивание по ключу. Оператор `del` также работает; он удаляет элемент с ключом, указанным в качестве индекса. Обратите внимание на вложение списка внутрь словаря в следующем примере (значение ключа 'ham'). Все типы данных коллекций в Python допускают произвольное вложение друг в друга:

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
>>> D['ham'] = ['grill', 'bake', 'fry'] # Изменение элемента (значение - список)
```

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> del D['eggs'] # Удаление элемента
>>> D
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> D['brunch'] = 'Bacon' # Добавление нового элемента
>>> D
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

Как и в списках, присваивание по существующему индексу в словаре изменяет ассоциированное с индексом значение. Однако в отличие от списков всякий раз, когда выполняется присваивание по *новому* ключу словаря (по которому присваивание еще не производилось), в словаре создается новый элемент, что в предыдущем примере происходило для ключа 'brunch'. Для списков это не работает, т.к. разрешено присваивать только по существующим смещениям в списках – Python считает смещение, выходящее за конец списка, запрещенным и генерирует ошибку. Для расширения списка понадобится использовать инструменты вроде метода `append` или присваивания по срезу.

Дополнительные словарные методы

Словарные методы предлагают разнообразные инструменты, специфичные для типа. Например, словарные методы `values` и `items` возвращают соответственно значения словаря и кортежи с парами (*ключ, значение*); вместе с методом `keys` их удобно применять в циклах, которые нужны для прохода по элементам по очереди (примеры таких циклов начнут приводиться в следующем разделе). Как и `keys`, в Python 3.X указанные два метода возвращают *итерируемые* объекты, поэтому они помещены в вызов `list` для сбора всех значений с целью отображения:

```
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> list(D.values())
[3, 2, 1]
>>> list(D.items())
[('eggs', 3), ('spam', 2), ('ham', 1)]
```

В реальных программах, которые накапливают данные в ходе своего выполнения, часто невозможно предугадать, что будет находиться в словаре, до запуска программы и тем более при написании кода. Извлечение по несуществующему ключу обычно является ошибкой, но метод `get` возвращает стандартное значение – `None` или переданное ему значение, если ключ не существует. Таким способом легко предоставить стандартное значение для ключа, который не существует, и избежать ошибки, связанной с отсутствующим ключом, когда предугадать заранее содержимое в программе невозможно:

```
>>> D.get('spam') # Ключ присутствует
2
>>> print(D.get('toast')) # Ключ отсутствует
None
>>> D.get('toast', 88)
88
```

Метод `update` предлагает своего рода конкатенацию для словарей, хотя он ничего не делает в плане упорядочения слева направо (в словарях такое понятие отсутствует).

Метод `update` *объединяет* ключи и значения одного словаря с ключами и значениями другого словаря, вслепую переписывая значения того же самого ключа, если возник конфликт:

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
>>> D2 = {'toast':4, 'muffin':5}    # Объединение словарей
>>> D.update(D2)
>>> D
{'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2, 'ham': 1}
```

Обратите внимание на перемешанный порядок следования ключей в результирующем словаре; именно так работают словари. Наконец, словарный метод `pop` удаляет ключ из словаря и возвращает значение, которое с ним было ассоциировано. Он похож на списковый метод `pop`, но вместо необязательной позиции принимает ключ:

```
# Извлечение из словаря по ключу
>>> D
{'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2, 'ham': 1}
>>> D.pop('muffin')
5
>>> D.pop('toast')    # Удаление ключа и возвращение связанного с ним значения
4
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}

# Извлечение из списка по позиции
>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop()           # Удаление и возвращение из конца
'dd'
>>> L
['aa', 'bb', 'cc']
>>> L.pop(1)          # Удаление по указанной позиции
'bb'
>>> L
['aa', 'cc']
```

Словари также предоставляют метод `sort`; мы вернемся к нему в главе 9, т.к. это способ избегания потенциальных побочных эффектов разделяемых ссылок в одном словаре. В действительности словари имеют больше методов, чем перечислено в табл. 8.2; полный список можно получить в руководстве по библиотеке Python, посредством вызовов `dir` и `help` либо из других источников.



Порядок в ваших словарях может отличаться. Не беспокойтесь, если ваши словари выводятся в порядке, который отличается от показанного здесь. Как уже упоминалось, порядок следования ключей произволен и может варьироваться в зависимости от выпуска, платформы и интерактивного сеанса в версии Python 3.3 (и вполне возможно даже от дня недели и фазы луны!).

Большинство примеров использования словарей в книге отражают упорядочение ключей в Python 3.X, но оно будет другим как после, так и до версии Python 3.0. Порядок следования ключей в вашей версии Python может отличаться, однако это вовсе не повод беспокоиться: словари обрабатываются по ключам, а не по позициям. Программы не должны полагаться на произвольный порядок ключей в словарях, даже приведенный в книгах.

В стандартной библиотеке Python имеются типы расширений, поддерживающие порядок вставки среди своих ключей (тип `OrderedDict` в модуле `collections`), но они являются гибридами, с которыми связаны накладные расходы в виде добавочного пространства памяти и более низкой скорости работы, обусловленные наличием дополнительных возможностей, а не подлинными словарями. В них ключи хранятся внутри связного списка для поддержки операций над последовательностями.

Как будет показано в главе 9, в модуле `collections` также реализован тип `namedtuple`, делающий возможным доступ к элементам кортежа как по именам атрибутов, так и по последовательным позициям – своего рода гибрид кортежа/класса/словаря с дополнительными шагами обработки, который в любом случае не входит в состав основных типов объектов. Указанные и многие другие типы расширений подробно описаны в руководстве по библиотеке Python.

Пример: база данных о фильмах

Давайте рассмотрим более реалистичный пример словаря. В следующем примере создается простая хранящаяся в памяти база данных о фильмах Monty Python, имеющая вид таблицы, которая отображает *годы выхода фильмов* (ключи) на *названия фильмов* (значения). Названия фильмов извлекаются путем индексации по строкам с *годами выхода*:

```
>>> table = {'1975': 'Holy Grail', # Ключ: Значение
...           '1979': 'Life of Brian',
...           '1983': 'The Meaning of Life'}
>>>
>>> year = '1983'
>>> movie = table[year]                  # словарь [Ключ] => Значение
>>> movie
'The Meaning of Life'
>>> for year in table:                 # То же, что и for year in table.keys()
...     print(year + '\t' + table[year])
...
1979 Life of Brian
1975 Holy Grail
1983 The Meaning of Life
```

В последней команде применяется цикл `for`, который был кратко представлен в главе 4, но пока еще подробно не раскрывался. Если вы не знакомы с циклами `for`, то знайте, что команда просто проходит по всем ключам в таблице и выводит разделенный табуляциями список ключей и связанных с ними значений. Циклы `for` обсуждаются в главе 13.

Словари не являются последовательностями как списки и строки, но если нужно пройти по элементам в словаре, то это легко – вызов словарного метода `keys` возвращает все сохраненные *ключи*, по которым можно проходить с помощью `for`. При необходимости внутри цикла `for` можно получать *значение* посредством индексации по *ключу*, что и было сделано в коде.

В действительности Python также дает возможность проходить по списку ключей словаря, фактически не вызывая метод `keys` в большинстве циклов `for`. Для любого словаря `D` конструкция `for key in D` работает точно так же, как полный оператор `for key in D.keys()`. На самом деле это всего лишь еще один пример упоминавших-

ся ранее *итераторов*, которые позволяют операции членства `in` работать также со словарями; итераторы будут подробно описаны позже в книге.

Обзор: отображение значений на ключи

Обратите внимание, что предыдущая таблица отображает годы на названия, но не наоборот. Если вы хотите отображать в противоположном направлении, т.е. названия на годы, то можете либо организовать словарь по-другому, либо использовать методы вроде `items`, который дает последовательности с возможностью поиска, хотя их эффективное применение требует большего объема знаний, чем имеется в текущий момент:

```
>>> table = {'Holy Grail': '1975',           # Ключ=>Значение (название=>год)
...             'Life of Brian': '1979',
...             'The Meaning of Life': '1983'}
>>>
>>> table['Holy Grail']
'1975'
>>> list(table.items())                  # Значение=>Ключ (год=>название)
[('The Meaning of Life', '1983'), ('Holy Grail', '1975'), ('Life of Brian', '1979')]
>>> [title for (title, year) in table.items() if year == '1975']
['Holy Grail']
```

Последняя команда отчасти является иллюстрацией синтаксиса *включения*, который был введен в главе 4 и полностью раскрывается в главе 14. Он просматривает пары кортежей (*ключ, значение*) словаря, возвращенные методом `items`, выбирая ключи с указанным значением. Совокупным эффектом будет индексация в *обратном направлении*, т.е. от значения к ключу, а не от ключа к значению. Такая индексация удобна, когда данные желательно сохранить только раз и отображать в обратном направлении лишь редко (подобного рода поиск в последовательностях обычно выполняется гораздо медленнее, чем прямая индексация по ключу).

На самом деле, хотя словари по своей природе отображают ключи на значения однодirectional, существует множество способов отображения значений обратно на ключи посредством чуть более обобщенного кода:

```
>>> K = 'Holy Grail'
>>> table[K]                         # Ключ=>Значение (нормальное использование)
'1975'
>>> V = '1975'
>>> [key for (key, value) in table.items() if value == V]    # Значение=>Ключ
['Holy Grail']
>>> [key for key in table.keys() if table[key] == V]        # То же самое
['Holy Grail']
```

Имейте в виду, что последние две команды возвращают *список* названий: в словарях есть только *одно* значение на ключ, но может быть *много* ключей на значение. Заданное значение может быть сохранено под множеством ключей (выдавая много ключей на значение), и само значение может быть коллекцией (поддерживая много значений на ключ). Также дождитесь функции обращения словаря в примере `mapattr.py` из главы 32 – код, который непременно растянул бы настоящий обзор до критических размеров, будь он включен здесь. Преследуя цели текущей главы, давайте продолжим исследование основ словарей.

Замечания по использованию словарей

Словари окажутся довольно-таки прямолинейными инструментами, как только вы освоитесь с ними, но ниже приведено несколько дополнительных указаний и памяток, о которых следует знать, имея дело со словарями.

- Операции над последовательностями не работают. Словари являются отображениями, не последовательностями; поскольку понятие порядка среди их элементов отсутствует, вещи вроде конкатенации (упорядоченное объединение) и нарезания (извлечение непрерывной секции) попросту неприменимы. В действительности Python инициирует ошибку во время выполнения кода, в котором предпринимается попытка сделать что-то подобное.
- Присваивание по новым индексам добавляет элементы. Ключи могут создаваться при написании словарного литерала (встраиваться в код самого литерала) либо во время присваивания значений новым ключам существующего объекта словаря на индивидуальной основе. Конечный результат будет таким же.
- Ключи не обязаны всегда быть строками. До сих пор в примерах в качестве ключей использовались строки, но допускается применять любые неизменяемые объекты. Скажем, вы можете использовать для ключей целые числа, что сделает словарь очень похожим на список (во всяком случае, при индексировании). Кортежи также могут применяться как ключи словаря, позволяя составным значениям ключей, таким как даты и IP-адреса, иметь ассоциированные значения. Ключами могут выступать объекты классов, определяемых пользователем (обсуждаются в части VI), при наличии в них подходящих протокольных методов. Грубо говоря, им необходимо сообщить Python о том, что их значения являются "хешируемыми" и потому изменяться не будут, т.к. иначе они оказались бы бесполезными как фиксированные ключи. Изменяемые объекты вроде списков, множеств и других словарей не могут быть ключами, но разрешены в качестве значений.

Использование словарей для имитации гибких списков: целочисленные ключи

Последний пункт в списке выше достаточно важен, чтобы продемонстрировать его на нескольких примерах. В случае применения списков присваивать по смещению, попадающему за конец списка, запрещено:

```
>>> L = []
>>> L[99] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка индекса: присваивание в списке по индексу,
выходящему за допустимые пределы
```

Несмотря на то что вы можете использовать повторение для предварительного выделения такого объема памяти, который необходим для списка (например, `[0]*100`), что-то похожее можно делать со словарями, не требующими такого выделения памяти. За счет применения целочисленных ключей словари способны имитировать списки, которые увеличиваются при присваивании по смещению:

```
>>> D = {}
>>> D[99] = 'spam'
>>> D[99]
'spam'
>>> D
{99: 'spam'}
```

Здесь все выглядит так, как если бы объект D был 100-элементным списком, но на самом деле он представляет собой словарь с единственным элементом; значением ключа 99 является строка 'spam'. Получать доступ к этой структуре можно через смещения почти как в списке, захватывая несуществующие ключи в вызовах `get` или проверках `in`, когда требуется, но выделять память для всех позиций, которым могут присваиваться значения в будущем, не понадобится. При таком использовании словари выглядят более гибкими эквивалентами списков.

В качестве еще одного примера мы можем задействовать целочисленные ключи в коде ранее рассмотренной базы данных о фильмах, чтобы избежать помещения в кавычки годов, хотя и ценой снижения выразительности (ключи не могут содержать нецифровые символы):

```
>>> table = {1975: 'Holy Grail',
...             1979: 'Life of Brian',      # Ключи являются целыми числами,
...                   #    не строками
...             1983: 'The Meaning of Life'}
>>> table[1975]
'Holy Grail'
>>> list(table.items())
[(1979, 'Life of Brian'), (1983, 'The Meaning of Life'), (1975, 'Holy Grail')]
```

Использование словарей для разреженных структур данных: ключи в форме кортежей

Похожим образом словарные ключи также обычно применяются для реализации *разреженных структур данных* – скажем, многомерных массивов, где значения хранятся лишь в нескольких позициях:

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4    # ; отделяет операторы: см. главу 10
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

Здесь мы используем словарь для представления трехмерного массива, который пуст за исключением двух позиций (2,3,4) и (7,8,9). Ключи являются *кортежами*, которые регистрируют координаты непустых ячеек. Вместо выделения памяти под крупную и по большей части пустую трехмерную матрицу для хранения этих значений, мы можем применять простой двухэлементный словарь. В такой схеме доступ пустой ячейке приводит к генерации исключения несуществующего ключа, потому что пустые ячейки физически не хранятся:

```
>>> Matrix[(2,3,6)]
Traceback (most recent call last):
File "<stdin>", line 1, in ?
```

```
KeyError: (2, 3, 6)
Трассировка (самый последний вызов указан последним):
Файл <stdin>, строка 1, в ?
Ошибка ключа: (2, 3, 6)
```

Избегание ошибок, связанных с отсутствующими ключами

Ошибки извлечения несуществующих ключей распространены в разреженных матрицах, но вряд ли кто-то захочет, чтобы они вызывали прекращение работы программы. Имеются, по меньшей мере, три способа заполнения стандартным значением вместо получения такого сообщения об ошибке — можно заранее проверять ключи в операторах `if`, использовать оператор `try` для явного перехвата и восстановления после исключения или применять показанный ранее словарный метод `get` для представления стандартного значения ключей, которые не существуют. Рассмотрим первые два способа в рамках обзора синтаксиса операторов, который мы начнем исследовать в главе 10:

```
>>> if (2, 3, 6) in Matrix:          # Проверка наличия ключа перед извлечением
...     print(Matrix[(2, 3, 6)])      # См. главы 10 и 12 для получения
...                               # сведений об if/else
...
... else:
...     print(0)
...
0
>>> try:
...     print(Matrix[(2, 3, 6)])      # Попытка индексации
... except KeyError:
...     print(0)                      # См. главы 10 и 34 для получения
...                               # сведений о try/except
...
0
>>> Matrix.get((2, 3, 4), 0)        # Существует: извлечь и возвратить
88
>>> Matrix.get((2, 3, 6), 0)        # Не существует: использовать
...                               # стандартный аргумент
0
```

Метод `get` самый лаконичный в плане требований написания кода, но операторы `if` и `try` гораздо более универсальны в отношении свободы действий; мы начнем обсуждать их в главе 10.

Вложение словарей

Как видите, словари способны исполнять многие роли в Python. В целом они могут заменять поисковые структуры данных (поскольку индексация по ключу является операцией поиска) и представлять множество видов структурированной информации. Например, словари — это один из многих способов описания свойств элемента в предметной области программы; т.е. они в состоянии служить тем же целям, что и “записи” или “структуры” в других языках.

В следующем примере заполняется словарь, описывающий гипотетическое лицо, путем присваивания по новым ключам со временем:

```
>>> rec = {}
>>> rec['name'] = 'Bob'
>>> rec['age'] = 40.5
```

```
>>> rec['job'] = 'developer/manager'  
>>>  
>>> print(rec['name'])  
Bob
```

Встроенные типы данных Python, особенно когда они вкладываются, позволяют легко представлять *структурированную* информацию. В показанном ниже примере словарь снова используется для фиксации свойств объекта, но все свойства записываются сразу (вместо отдельных присваиваний по каждому ключу), а значения структурированных свойств представляются с помощью вложенных списка и словаря:

```
>>> rec = {'name': 'Bob',  
...         'jobs': ['developer', 'manager'],  
...         'web': 'www.bobs.org/~Bob',  
...         'home': {'state': 'Overworked', 'zip': 12345}}
```

Чтобы извлечь компоненты вложенных объектов, нужно просто выстроить в цепочку операции индексирования:

```
>>> rec['name']  
'Bob'  
>>> rec['jobs']  
['developer', 'manager']  
>>> rec['jobs'][1]  
'manager'  
>>> rec['home']['zip']  
12345
```

Несмотря на то что в части VI станет известно, что *классы* (группирующие данные и логику) способны более эффективно исполнять роль записей, словари являются легким в применении инструментом для удовлетворения более простых требований. Дополнительные варианты представлений обсуждаются во врезке “Что потребует внимания: словари или списки” далее в главе, а также при рассмотрении расширения кортежей в главе 9 и классов в главе 27.

Также обратите внимание, что хотя здесь мы сосредоточились на одиночной “записи” с вложенными данными, нет никаких причин, по которым мы не могли бы вложить саму запись в более крупную коллекцию *базы данных*, представленную в виде списка или словаря. Тем не менее, в реальных программах роль контейнера верхнего уровня часто исполняет интерфейс к внешнему файлу или базе данных (показанные далее фрагменты выводят двухэлементный список работ сотрудника Bob, если они выполняются в реальном времени и получают еще одну структуру записи):

```
db = []  
db.append(rec)      # "База данных" в форме списка  
db.append(other)  
db[0]['jobs']  
  
db = {}  
db['bob'] = rec    # "База данных" в форме словаря  
db['sue'] = other  
db['bob']['jobs']
```

Позже в книге мы встретим инструменты, такие как модуль *shelve*, который работает во многом аналогично, но автоматически сопоставляет объекты с файлами, чтобы обеспечить их постоянство (дополнительные сведения приведены во врезке “Что потребует внимания: словарные интерфейсы” в конце главы).

Другие способы создания словарей

Наконец, следует отметить, что поскольку словари настолько полезны, со временем появилось больше способов их построения. Например, в Python 2.3 и новых версиях последние два вызова конструктора `dict` (в действительности имени типа) дают такой же эффект, как формы литературного выражения и присваивания по ключам перед ними:

```
{'name': 'Bob', 'age': 40}      # Традиционное литературное выражение  
D = {}                          # Динамическое присваивание по ключам  
D['name'] = 'Bob'  
D['age'] = 40  
  
dict(name='Bob', age=40)        # Форма dict с ключевыми аргументами  
dict([('name', 'Bob'), ('age', 40)]) # Форма dict с кортежами "ключ/значение"
```

Все четыре формы создают один и тот же словарь с двумя ключами, но они удобны в разных обстоятельствах.

- Первая форма удобна, если вы в состоянии записать словарь заранее.
- Вторая форма используется, когда словарь необходимо создавать на лету по одному полю за раз.
- Третья форма сопряжена с меньшим объемом набора, чем первая, но требует, чтобы все ключи были строками.
- Четвертая форма применяется, если ключи и значения нужно накапливать как последовательности во время выполнения.

Мы встречали ключевые аргументы ранее при исследовании сортировки; третья форма, продемонстрированная в примере выше, стала особенно популярной в современном коде Python, т.к. ее синтаксис короче (и потому меньше шансов допустить ошибки). Как можно было предположить из табл. 8.2, последняя форма в листинге также часто используется в сочетании с функцией `zip` для объединения отдельных списков ключей и значений, получаемых динамически во время выполнения (скажем, в результате разбора столбцов файла данных):

```
dict(zip(keyslist, valueslist)) # Форма с упакованными кортежами "ключ/значение"
```

Упакованные словарные ключи более подробно обсуждаются в следующем разделе. При условии, что значения, ассоциированные со всеми ключами, первоначально одинаковы, словарь можно также создать с помощью следующей специальной формы — просто передавая список ключей и начальную величину для всех значений (по умолчанию принимается `None`):

```
>>> dict.fromkeys(['a', 'b'], 0)  
{'a': 0, 'b': 0}
```

Хотя на данном этапе вашей карьеры программиста на Python вы могли бы обойтись только литературными выражениями и присваиваниями по ключу, вы наверняка найдете применение для всех форм создания словарей, когда приступите к написанию реалистичных, гибких и динамических программ.

В листингах текущего раздела иллюстрируются разнообразные способы создания словарей в версиях Python 2.X и Python 3.X. Однако существует еще один способ создания словарей, доступный только в Python 3.X и Python 2.7: выражение `включения словаря`. Чтобы посмотреть, как выглядит такая форма, понадобится перейти к следующему и последнему разделу главы.

Что потребует внимания: словари или списки

При таком богатом арсенале объектов основных типов в Python некоторые читатели могут ломать голову над выбором между списками и словарями. Говоря кратко, хотя оба типа являются гибкими коллекциями других объектов, в списках элементы присваиваются по *позициям*, а в словарях – по мнемоническим *ключам*. Из-за этого данные словарей часто несут больше смысла для людей, читающих код. Например, вложенная списковая структура из третьей строки в табл. 8.1 могла бы также использоватьсь для представления записи:

```
>>> L = ['Bob', 40.5, ['dev', 'mgr']]    # "Запись" на основе списка
>>> L[0]
'Bob'
>>> L[1]                                # Позиции/числа для полей
40.5
>>> L[2][1]
'mgr'
```

Для определенных видов данных доступ по позиции списка имеет смысл – скажем, в перечнях сотрудников компании, файлов каталога или числовых матрицах. Но приведенная далее символьическая запись может быть более выразительно записана как словарь согласно второй строке в табл. 8.2 с заменой позиций полей помеченными полями (она похожа на запись из главы 4):

```
>>> D = {'name': 'Bob', 'age': 40.5, 'jobs': ['dev', 'mgr']}
>>> D['name']
'Bob'
>>> D['age']      # "Запись" на основе словаря
40.5
>>> D['jobs'][1]   # Имена выразительнее чисел
'mgr'
```

Для разнообразия вот та же самая запись в форме с ключевыми словами, которая некоторым людям может показаться даже еще более читабельной:

```
>>> D = dict(name='Bob', age=40.5, jobs=['dev', 'mgr'])
>>> D['name']
'Bob'
>>> D['jobs'].remove('mgr')
>>> D
{'jobs': ['dev'], 'age': 40.5, 'name': 'Bob'}
```

На практике словари обычно лучше подходят для данных с помеченными компонентами, а также для структур, которые могут извлечь преимущество от быстрого и прямого поиска по имени вместо медленного линейного поиска. Как было показано, они могут быть более пригодны для разреженных коллекций и коллекций, увеличивающихся в произвольных позициях.

Программистам на Python также доступны *множества*, обсуждавшиеся в главе 5, которые очень похожи на ключи словарей баз значений; они не отображают ключи на значения, но часто могут применяться как словари для быстрого поиска, когда связанные значения отсутствуют, особенно в процедурах поиска:

```
>>> D = {}
>>> D['state1'] = True    # Словарь посещенных состояний
>>> 'state1' in D
True
>>> S = set()
>>> S.add('state1')       # То же самое, но с использованием множества
>>> 'state1' in S
True
```

Придание других форм этому представлению записи будет продолжено в следующей главе, где мы сравним *кортежи* и *именованные кортежи* со словарями в такой роли, и в главе 27, где будет показано, как в общую картину вписываются *классы*, определяемые пользователем, которые объединяют данные и обрабатывающую их логику.

Изменения в словарях в Python 3.X и 2.7

До сих пор внимание в главе было сосредоточено на основах словарей, охватывающих многие выпуски, но в версиях Python 3.X функциональность словаря видоизменилась. Если вы используете код Python 2.X, тогда можете столкнуться с инструментами словарей, которые либо ведут себя по-другому, либо вообще отсутствуют в Python 3.X. Кроме того, программисты на Python 3.X могут обращаться к дополнительным инструментам словарей, недоступных в Python 2.X за исключением обратных переносов в Python 2.7. В частности, словари в *Python 3.X*:

- поддерживают новое выражение включения словарей, близкий родственник списковым включениям и включениям множеств;
- возвращают итерируемые представления, подобные множествам, вместо списков для методов `D.keys`, `D.values` и `D.items`;
- из-за предыдущего пункта требуют новых стилей написания кода для просмотра по сортированным ключам;
- больше не поддерживают сравнения по относительной величине напрямую – взамен придется сравнивать вручную;
- больше не имеют метода `D.has_key` – взамен необходимо применять проверку членства `in`.

Из-за более поздних обратных переносов из Python 3.X словари в *Python 2.7* (но не в предшествующих версиях Python 2.X):

- поддерживают первый пункт предыдущего списка – включения словарей – как обратный перенос из Python 3.X;
- поддерживают второй пункт предыдущего списка – итерируемые представления, подобные множествам, – но посредством методов со специальными именами `D.viewkeys`, `D.viewvalues` и `D.viewitems`; методы, не имеющие дела с представлениями, возвращают списки, как и ранее.

По причине такого частичного наложения некоторые материалы в настоящем разделе относятся к Python 3.X и Python 2.7, но приведены здесь в контексте расширений Python 3.X из-за своего происхождения. Имея это в виду, давайте посмотрим, что нового в словарях Python 3.X и 2.7.

Включения словарей в Python 3.X and 2.7

Как упоминалось в конце предыдущего раздела, словари в Python 3.X и 2.7 можно создавать также с помощью включений словарей. Подобно включениям множеств, которые мы встречали в главе 5, включения словарей доступны только в Python 3.X и 2.7 (не в Python 2.6 и предшествующих версиях). Как и издавна существующие списковые включения, кратко упоминаемые в главе 4 и ранее в текущей главе, они выполняют подразумеваемый цикл, накапливая на каждой итерации результаты “ключ/значение” и используя их для наполнения нового словаря. Переменная цикла позволяет включению попутно использовать значения итерации цикла.

В качестве иллюстрации стандартный способ динамической инициализации словаря в Python 2.X и 3.X предусматривает объединение его ключей и значений посредством zip с передачей результата вызову dict. Встроенная функция zip – это прием, который позволяет создавать словарь из списков ключей и значений подобным образом; если вы не в состоянии спрогнозировать набор ключей и значений в своем коде, тогда всегда можете построить их как списки и упаковать вместе. Функция zip будет подробно рассматриваться в главах 13 и 14 после исследования операторов; в Python 3.X она возвращает итерируемый объект, поэтому здесь для отображения результатов мы обязаны поместить ее внутрь вызова list, но ее базовое применение в других обстоятельствах прямолинейно:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3])) # Упаковка вместе ключей и значений
[('a', 1), ('b', 2), ('c', 3)]
>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))    # Создание словаря
                                                # из результата zip
>>> D
{'b': 2, 'c': 3, 'a': 1}
```

Тем не менее, в Python 3.X и 2.7 того же эффекта можно достичь посредством выражения включения словаря. Ниже строится новый словарь с парами “ключ/значение” для каждой такой пары в результате zip (код Python читается почти так же, как описание на естественном языке, но чуть более формально):

```
>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
>>> D
{'b': 2, 'c': 3, 'a': 1}
```

Фактически в этом случае включения требуют большего объема кода, но они более универсальны, чем вытекает из приведенного примера – их допускается использовать также для отображения одиночного потока значений на словари, а ключи могут вычисляться с помощью выражений, как и значения:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}           # Или range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}
>>> D = {c: c * 4 for c in 'SPAM'}      # Проход в цикле по любому
                                            # итерируемому объекту
>>> D
{'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM'}
>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
>>> D
{'eggs': 'EGGS!', 'spam': 'SPAM!', 'ham': 'HAM!}'}
```

Включения словарей также удобны для инициализации словарей из списков ключей во многом таким же способом, как метод fromkeys, который был представлен в конце предыдущего раздела:

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0) # Инициализация словаря из ключей
>>> D
{'b': 0, 'c': 0, 'a': 0}
>>> D = {k:0 for k in ['a', 'b', 'c']} # То же самое, но посредством включения
>>> D
{'b': 0, 'c': 0, 'a': 0}
>>> D = dict.fromkeys('spam') # Другие итерируемые объекты, стандартное значение
>>> D
{'s': None, 'p': None, 'a': None, 'm': None}
```

```
>>> D = {k: None for k in 'spam'}
>>> D
{'s': None, 'p': None, 'a': None, 'm': None}
```

Как и связанные инструменты, включения словарей поддерживают дополнительный синтаксис, который здесь не показан, в том числе вложенные циклы и конструкции `if`. К сожалению, для полного понимания включений словарей необходимо также больше знать об операторах и концепциях итерации в Python, а в данный момент информации недостаточно, чтобы хорошо в них разобраться. В главах 14 и 20 будут более подробно обсуждаться все разновидности включений (списков, множеств, словарей, а также генераторы), поэтому мы откладываем дальнейшие детали до тех пор. Встроенная функция `zip`, применяемая в текущем разделе, будет рассматриваться в главе 13 во время исследования циклов `for`.

Словарные представления в Python 3.X (и в Python 2.7 через новые методы)

В Python 3.X словарные методы `keys`, `values` и `items` возвращают *объекты представлений*, тогда как в Python 2.X они возвращают фактические результирующие списки. Такая функциональность доступна и в Python 2.7, но в обличье методов со специальными, отличающимися именами, которые были указаны в начале раздела (обычные методы Python 2.7 по-прежнему возвращают простые списки во избежание нарушения работы существующего кода Python 2.X). По этой причине в данном разделе они позиционируются как средство Python 3.X.

Объекты представлений являются *итерируемыми*, т.е. они генерируют результирующие элементы по одному за раз, а не производят сразу весь результирующий список в памяти. Помимо того, что объекты представлений итерируемые, они также предохраняют первоначальный порядок следования компонентов в словаре, отражают будущие изменения, вносимые в словарь, и способны поддерживать операции над множествами. С другой стороны, поскольку объекты представлений – не списки, они не поддерживают напрямую операции, подобные индексации или списковому методу `sort`. Кроме того, при выводе они не отображают свои элементы как нормальный список (начиная с версии Python 3.1, объекты представлений выводят свои компоненты, но не в виде списка, что отличается от Python 2.X).

Понятие итерируемых объектов будет более формально обсуждаться в главе 14, но здесь достаточно знать, что при желании применить списковые операции или отобразить их значения результаты методов `keys`, `values` и `items` понадобится прогнать через встроенную функцию `list`. Вот пример для Python 3.3 (вывод в другой версии может слегка отличаться):

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> K = D.keys()      # В Python 3.X создается объект представления, не список
>>> K
dict_keys(['b', 'c', 'a'])
>>> list(K) # Принудительно создать реальный список в Python 3.X, если нужно
['b', 'c', 'a']
>>> V = D.values()    # То же самое для представлений значений и элементов
>>> V
dict_values([2, 3, 1])
>>> list(V)
[2, 3, 1]
```

```
>>> D.items()
dict_items([('b', 2), ('c', 3), ('a', 1)])
>>> list(D.items())
[('b', 2), ('c', 3), ('a', 1)]
>>> K[0]          # Без преобразования списковые операции терпят неудачу
TypeError: 'dict_keys' object does not support indexing
Ошибка типа: объект dict_keys не поддерживает индексацию
>>> list(K)[0]
'b'
```

Помимо отображения результатов в интерактивной подсказке вам вряд ли часто придется обращать внимание на такое изменение, потому что циклические конструкции в Python автоматически заставляют итерируемые объекты выдавать по одному результату на каждой итерации:

```
>>> for k in D.keys(): print(k)      # В циклах итераторы используются
автоматически
...
b
c
a
```

Вдобавок сами словари Python 3.X по-прежнему имеют итераторы, которые возвращают последовательные ключи – как и в Python 2.X, часто нет необходимости в прямом вызове keys:

```
>>> for key in D: print(key)        # По-прежнему нет нужды вызывать keys
# для выполнения итерации
...
b
c
a
```

Однако в отличие от списковых результатов Python 2.X словарные представления в Python 3.X не являются застывшими – они динамически отражают будущие изменения, вносимые в словарь после создания объекта представления:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> K = D.keys()
>>> V = D.values()
>>> list(K)          # Представление поддерживает тот же порядок, что и словарь
['b', 'c', 'a']
>>> list(V)
[2, 3, 1]
>>> del D['b']      # Изменение словаря на месте
>>> D
{'c': 3, 'a': 1}
>>> list(K)          # Изменение отражается в любых текущих объектах представлений
['c', 'a']
>>> list(V)          # В Python 2.X это не так! Списки отсоединены от словаря
[3, 1]
```

Словарные представления и множества

Также в отличие от списковых результатов Python 2.X объекты представлений Python 3.X, возвращаемые методом `keys`, *подобны множествам* и поддерживают распространенные операции над множествами, такие как пересечение и объединение; представления `values` не подобны множествам, но результаты `items` подобны, если их пары (`ключ, значение`) уникальны и хешируемы (*неизменяемы*). Учитывая тот факт, что множества ведут себя очень похоже на словари без значений (и даже могут записываться в фигурных скобках как словари в Python 3.X/2.7), мы имеем дело с логической симметрией. Согласно главе 5 элементы множества неупорядочены, уникальны и неизменяемы в точности как ключи словаря.

Вот как выглядят представления ключей, когда используются в операциях над множествами (здесь продолжается сеанс из предыдущего раздела); представления значений словаря никогда не будут подобными множествам, потому что их элементы не обязательно уникальны или неизменяемы:

```
>>> K, V
(dict_keys(['c', 'a']), dict_values([3, 1]))
>>> K | {'x': 4} #Представления ключей (и некоторых элементов) подобны множествам
{'c', 'x', 'a'}
>>> V & {'x': 4}
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'
Ошибка типа: неподдерживаемые типы operandов для &: dict_values и dict
>>> V & {'x': 4}.values()
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_values'
Ошибка типа: неподдерживаемые типы operandов для &: dict_values и dict_values
```

В операциях над множествами представления могут смешиваться с другими представлениями, множествами и словарями; в таком контексте словари трактуются также, как их представления ключей:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D.keys() & D.keys()          # Пересечение представлений ключей
{'b', 'c', 'a'}
>>> D.keys() & {'b'}            # Пересечение представления ключей и множества
{'b'}
>>> D.keys() & {'b': 1}         # Пересечение представления ключей и словаря
{'b'}
>>> D.keys() | {'b', 'c', 'd'} # Объединение представления ключей и множества
{'b', 'c', 'a', 'd'}
```

Представления элементов тоже подобны множествам, если они хешируемы, т.е. если содержат только неизменяемые объекты:

```
>>> D = {'a': 1}
>>> list(D.items()) # Представления элементов подобны множествам, если хешируемы
[('a', 1)]
>>> D.items() | D.keys()      # Объединение представления и представления
{('a', 1), 'a'}
>>> D.items() | D            # Словарь трактуется так же, как его ключи
{('a', 1), 'a'}
>>> D.items() | {('c', 3), ('d', 4)}      # Множество пар "ключ/значение"
{('d', 4), ('a', 1), ('c', 3)}
```

```
>>> dict(D.items() | {('c', 3), ('d', 4)}) # Словарь также принимает
                                                # итерируемые множества
{'c': 3, 'a': 1, 'd': 4}
```

Если необходимо вспомнить о продемонстрированных выше операциях, тогда обратитесь к описанию множеств в главе 5. В заключение взгляните на три кратких замечания, касающиеся работы со словарями в Python 3.X.

Сортировка ключей словаря в Python 3.X

Во-первых, поскольку метод `keys` не возвращает список в Python 3.X, традиционный шаблон кодирования для просмотра словаря по отсортированным ключам из Python 2.X в Python 3.X работать не будет:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'b': 2, 'c': 3, 'a': 1}

>>> Ks = D.keys()      # Сортировка объекта представления не работает!
>>> Ks.sort()
AttributeError: 'dict_keys' object has no attribute 'sort'
Ошибка атрибута: объект dict_keys не имеет атрибута sort
```

Чтобы решить проблему, в Python 3.X придется либо вручную выполнить преобразование в список, либо применить вызов встроенной функции `sorted` (кратко упоминаемой в главе 4 и раскрываемой в этой главе) на представлении ключей или на самом словаре:

```
>>> Ks = list(Ks)                      # Преобразовать в список и затем сортировать
>>> Ks.sort()
>>> for k in Ks: print(k, D[k]) # Python 2.X: опустите внешние скобки в print
...
a 1
b 2
c 3

>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> Ks = D.keys()                      # Или можно вызвать sorted на ключах
>>> for k in sorted(Ks): print(k, D[k]) # sorted принимает любой
                                            # итерируемый объект
                                            # sorted возвращает свой результат
...
a 1
b 2
c 3
```

Использование итератора ключей словаря вероятно предпочтительнее в Python 3.X и работает также в Python 2.X:

```
>>> D
{'b': 2, 'c': 3, 'a': 1}          # Еще лучше сортировать словарь напрямую
>>> for k in sorted(D): print(k, D[k]) # Итераторы словаря возвращают ключи
...
a 1
b 2
c 3
```

Сравнения по абсолютной величине больше не работают в Python 3.X

Во-вторых, хотя в Python 2.X словари можно сравнивать по абсолютной величине напрямую с помощью `<`, `>` и т.д., в Python 3.X это больше не работает. Тем не менее, его можно эмулировать путем сравнения отсортированных списков ключей вручную:

```
sorted(D1.items()) < sorted(D2.items())      # Похоже на D1 < D2 в Python 2.X
```

Однако проверки словарей на равенство (например, `D1 == D2`) в Python 3.X по-прежнему работают. Так как мы вернемся к данной теме ближе к концу следующей главы в контексте сравнений в целом, здесь мы не будем вдаваться в дальнейшие детали.

Метод `has_key` умер в Python 3.X: да здравствует операция `in`!

Наконец, в-третьих, широко применяемый словарный метод проверки наличия ключа `has_key` в Python 3.X исчез. Взамен необходимо использовать операцию членства `in` либо метод `get` с проверкой на равенство стандартному значению (из них операция `in` обычно предпочтительнее):

```
>>> D
{'b': 2, 'c': 3, 'a': 1}

>>> D.has_key('c')                      # Только Python 2.X: True/False
AttributeError: 'dict' object has no attribute 'has_key'
Ошибка атрибута: объект dict не имеет атрибута has_key

>>> 'c' in D                          # Требуется в Python 3.X
True
>>> 'x' in D                          # Теперь предпочтительнее в Python 2.X
False
>>> if 'c' in D: print('present', D['c'])    # Ветвление по результату
...
present 3

>>> print(D.get('c'))                  # Извлечение со стандартным значением
3
>>> print(D.get('x'))
None
>>> if D.get('c') != None: print('present', D['c'])    # Еще один вариант
...
present 3
```

Подводя итоги, ситуация со словарями в Python 3.X значительно изменилась. Если вы работаете в Python 2.X и заботитесь о совместимости с Python 3.X (или подозреваете, что когда-нибудь так случится), то ниже приведена пара указаний. Из изменений Python 3.X, которые рассматривались в настоящем разделе:

- первое (включения словарей) может записываться только в Python 3.X и Python 2.7;
- второе (словарные представления) может записываться только в Python 3.X и посредством методов со специальными именами в Python 2.7.

Тем не менее, последние три методики – функция `sorted`, ручные сравнения и операция `in` – в наши дни могут записываться в Python 2.X с целью облегчения перехода на Python 3.X в будущем.

Что потребует внимания: словарные интерфейсы

Словари – не просто удобный способ хранения информации по ключам в программах; дело в том, что ряд расширений Python также предлагает интерфейсы, которые выглядят похожими и работают аналогично словарям. Скажем, интерфейс Python к файлам DBM с доступом по ключу выглядит очень похожим на словарь, который должен открываться. Строки сохраняются и извлекаются с применением ключевых индексов:

```
import dbm          # В Python 2.X называется anydbm
file = dbm.open("filename")    # Связаться с файлом
file['key'] = 'data'        # Сохранить данные по ключу
data = file['key']         # Извлечь данные по ключу
```

В главе 28 вы увидите, что подобным способом можно также сохранять целые объекты Python, если вместо dbm в предыдущем коде указать shelve (база данных с доступом по ключу, которая хранит постоянные объекты Python, а не только строки). Для работы в Интернете поддержка Python сценариев CGI также имеет вид интерфейса, похожего на словари. Вызов cgi.FieldStorage выдает объект, подобный словарю, с записями для каждого поля ввода формы на клиентской веб-странице:

```
import cgi
form = cgi.FieldStorage()      # Разбор данных формы
if 'name' in form:
    showReply('Hello, ' + form['name'].value)
```

Хотя словари являются единственным основным типом отображения, все перечисленные интерфейсы представляют собой примеры отображений и поддерживают большинство тех же самых операций. Изучив словарные интерфейсы, вы обнаружите, что они применимы к разнообразным встроенным инструментам в Python.

Еще один сценарий использования словарей вы найдете в приведенном в главе 9 обзоре JSON – нейтрального к языкам формата данных, применяемого для баз данных и передачи данных. Словари, списки и вложенные их сочетания в Python могут почти полностью соответствовать записям в формате JSON и легко транслироваться в и из формальных строк текста JSON с помощью стандартного библиотечного модуля Python по имени json.

Резюме

В главе мы исследовали списковые и словарные типы – вероятно два наиболее распространенных, гибких и мощных типа коллекций, которые вы будете видеть и использовать в коде Python. Вы узнали, что списоковый тип поддерживает позиционно упорядоченные коллекции произвольных объектов и может свободно вкладываться, а также увеличиваться и уменьшаться по требованию. Словарный тип похож, но хранит элементы по ключам, а не по позициям, и не поддерживает среди своих элементов какого-либо надежного порядка слева направо. Списки и словари являются изменяемыми и потому поддерживают разнообразные операции изменения на месте, недоступные для строк: например, списки могут увеличиваться вызовами append, а словари – присваиванием по новым ключам.

В следующей главе мы закончим наше путешествие по типам основных объектов, рассмотрев кортежи и файлы. Затем мы перейдем к операторам, кодирующими логику, которая обрабатывает наши объекты, сделав еще один шаг в направлении написания завершенных программ. Однако прежде чем мы зайдемся такими темами, вот традиционные контрольные вопросы, которые помогут закрепить пройденный материал.

Проверьте свои знания: контрольные вопросы

1. Назовите два способа построения списка, содержащего пять целочисленных нулей.
2. Назовите два способа построения словаря с двумя ключами, 'a' и 'b', с каждым из которых ассоциировано значение 0.
3. Назовите четыре операции, которые изменяют списоковый объект на месте.
4. Назовите четыре операции, которые изменяют словарный объект на месте.
5. Почему вы можете использовать словарь вместо списка?

Проверьте свои знания: ответы

1. Список из пяти нулей создадут литеральное выражение вроде [0, 0, 0, 0, 0] и выражение повторения наподобие [0] * 5. На практике вы можете также построить такой список с помощью цикла, который начинает с пустого списка и на каждой итерации добавляет к нему 0 посредством L.append(0). Здесь могло бы подойти и списковое включение ([0 for i in range(5)]), но с ним связано больше работы, чем стоит делать в ответе на этот простой вопрос.
2. Желаемый словарь создадут литеральное выражение, такое как {'a': 0, 'b': 0}, или серия присваиваний вроде D = {}, D['a'] = 0 и D['b'] = 0. Вы также можете применять более новую и простую в наборе форму ключевого слова dict(a=0, b=0) или более гибкую форму последовательности "ключей/значений" dict([('a', 0), ('b', 0)]). Либо из-за того, что все значения одинаковы, вы можете использовать специальную форму dict.fromkeys('ab', 0). В Python 3.X и 2.7 вы также можете применять включение словаря: {k:0 for k in 'ab'}, хотя здесь оно снова может оказаться излишеством.
3. Методы append и extend увеличивают список на месте, методы sort и reverse упорядочивают и обращают списки, метод insert вставляет элемент по смещению, методы remove и pop удаляют элемент из списка по значению и по позиции, оператор del удаляет элемент или срез, а операторы присваивания по индексам и по срезам заменяют элемент или целую секцию. Для ответа на данный вопрос выберите любые четыре операции из перечисленных.
4. Словари изменяются главным образом путем присваивания по новому или существующему ключу, что создает либо изменяет запись в таблице, связанную с этим ключом. Кроме того, оператор del удаляет запись ключа, словарный метод update объединяет один словарь с другим на месте и метод D.pop(key) удаляет ключ и возвращает значение, которое он имел. У словарей также есть прочие более экзотические методы, не рассмотренные в главе, такие как setdefault; подробные сведения ищите в справочниках.
5. Словари обычно лучше, когда данные помечены (например, запись с именованными полями), а списки больше подходят для коллекций непомеченных данных (таких как все файлы в каталоге). Поиск в словаре, как правило, быстрее поиска в списке, хотя это может варьироваться в зависимости от программы.

Кортежи, файлы и все остальное

Настоящая глава завершает наш тур по типам основных объектов в Python исследованием *кортежа* – коллекции других объектов, которая не может изменяться, и *файла* – интерфейса к внешним файлам на компьютере. Как вы узнаете, кортеж является относительно простым объектом, который выполняет операции, большей частью вам уже знакомые по строкам и спискам. Файловый объект представляет собой широко используемый и полнофункциональный инструмент для обработки файлов на компьютере. Поскольку в программировании файлы вездесущи, приведенный здесь базовый обзор файлов дополняется более крупными примерами в последующих главах.

Глава также завершает часть II книги рассмотрением свойств, общих для всех типов основных объектов, которые мы встречали – понятий равенства, сравнений, копий объектов и т.д. Мы также бегло взглянем на другие типы объектов в инструментальном наборе Python, включая заполнитель `None` и гибрид `namedtuple`; как вы увидите, несмотря на то, что мы раскрыли все основные встроенные типы, история с объектами в Python шире, чем подразумевалось до сих пор. Наконец, в конце этой части книги мы рассмотрим набор распространенных ловушек, связанных с типами объектов, и предложим несколько упражнений, которые позволят вам поэкспериментировать с изученными идеями.



Вопросы, раскрываемые в главе – файлы. Как и в главе 7 о строках, рамки нашего обсуждения файлов будут ограничены фундаментальными основами, которые обязано знать большинство программистов на Python, включая новичков. Скажем, в главе 4 был дан краткий обзор *текстовых файлов Unicode*, но их полное раскрытие отложено до главы 37 как более сложная тема, собранная в часть книги для необязательного или отсроченного чтения.

Для целей этой главы мы предполагаем, что любые текстовые файлы будут кодироваться и декодироваться, как по умолчанию принято на вашей платформе, скажем, UTF-8 в Windows и ASCII или другая кодировка где-то еще (если вы не знаете, почему это важно, то вероятно и не должны двигаться вперед). Мы также допускаем, что имена файлов надлежащим образом закодированы в соответствие с платформой, хотя ради переносимости будем придерживаться имен ASCII.

Если текст и файлы Unicode являются критически важной темой для вас, тогда я рекомендую прочитать краткий обзор в главе 4 и продолжить их изучение в главе 37, освоив раскрываемые здесь основы файлов. Для всех остальных описание работы с файлами в настоящей главе будет применимо к обычным текстовым и двоичным файлам вроде тех, что будут встречаться далее в главе, а также к более сложным режимам обработки файлов, которые вы можете решить исследовать позже.

Кортежи

Последний тип коллекции в нашем исследовании – кортеж Python. Кортежи создают простые группы объектов. Они работают в точности как списки за исключением того, что не могут быть модифицированы на месте (неизменяемы), и обычно записываются как серия элементов в круглых, а не квадратных скобках. Невзирая на то, что кортежи не поддерживают столько методов, они разделяют большинство свойств со списками. Ниже кратко описаны основные характеристики кортежей.

Они являются упорядоченными коллекциями произвольных объектов

Подобно строкам и спискам кортежи представляют собой позиционно упорядоченные коллекции объектов (т.е. поддерживают в своем содержимом порядок слева направо); как и списки, они способны содержать объекты любых видов.

Они поддерживают доступ по смещению

Подобно строкам и спискам элементы в кортеже доступны по смещению (не по ключу); они поддерживают все операции доступа на основе смещения, такие как индексация и нарезание.

Они относятся к категории “неизменяемая последовательность”

Подобно строкам и спискам кортежи являются последовательностями; они поддерживают многие аналогичные операции. Однако, как и строки, кортежи неизменяемы; они не поддерживают любые операции изменения на месте, применимые к спискам.

Они имеют фиксированную длину, разнородны и допускают произвольно глубокое вложение

Поскольку кортежи неизменяемы, изменить размер кортежа без создания копии невозможно. С другой стороны, кортежи могут хранить объекты любого типа, включая другие составные объекты (например, списки, словари, другие кортежи), и потому поддерживают вложение на произвольную глубину.

Они представляют собой массивы ссылок на объекты

Подобно спискам кортежи лучше всего представлять себе как массивы ссылок на объекты; кортежи хранят точки доступа к другим объектам (ссылки), а индексация кортежа проходит относительно быстро.

В табл. 9.1 описаны распространенные операции над кортежами. Кортеж записывается как серия объектов (формально выражений, генерирующих объекты), разделенных запятыми и обычно заключенных в круглые скобки. Пустой кортеж – это просто пара круглых скобок, ничего не содержащая внутри.

Таблица 9.1. Распространенные литералы и операции над кортежами

Операция	Описание
()	Пустой кортеж
T = (0,)	Одноэлементный кортеж (не выражение)
T = (0, 'Ni', 1.2, 3)	Четырехэлементный кортеж
T = 0, 'Ni', 1.2, 3	Еще один четырехэлементный кортеж (такой же, как в предыдущей строке)
T = ('Bob', ('dev', 'mgr'))	Вложенные кортежи
T = tuple('spam')	Кортеж из элементов итерируемого объекта
T[i]	Индекс, индекс индекса, срез, длина
T[i][j]	
T[i:j]	
len(T)	
T1 + T2	Конкатенация, повторение
T*3	
for x in T: print(x)	Итерация, членство
'spam' in T	
[x ** 2 for x in T]	
T.index('Ni')	Методы в Python 2.6, 2.7 и 3.X: поиск, подсчет
T.count('Ni')	
namedtuple('Emp', ['name', 'jobs'])	Тип расширения именованного кортежа

Кортежи в действии

Как обычно, для исследования кортежей в работе запустим интерактивный сеанс. Обратите внимание в табл. 9.1, что кортежи не располагают всеми методами, которые есть у списков (скажем, вызов append не заработал бы). Тем не менее, они поддерживают обычные операции над последовательностями, которые мы видели для строк и списков:

```
>>> (1, 2) + (3, 4)      # Конкатенация
(1, 2, 3, 4)
>>> (1, 2) * 4          # Повторение
(1, 2, 1, 2, 1, 2, 1, 2)
>>> T = (1, 2, 3, 4)    # Индексация, нарезание
>>> T[0], T[1:3]
(1, (2, 3))
```

Особенности синтаксиса кортежей: запятые и круглые скобки

Вторая и четвертая строки в табл. 9.1 заслуживают чуть более подробных объяснений. Поскольку в круглые скобки могут заключаться также и выражения (см. главу 5), нужно предпринять что-то специальное для уведомления Python в ситуации, когда одиночный объект в круглых скобках является кортежем, а не простым выражением. Если в действительности вам необходим одноэлементный кортеж, тогда укажите после элемента и перед закрывающей скобкой хвостовую запятую:

```
>>> x = (40)      # Целое число!
>>> x
40
>>> y = (40,)    # Кортеж, содержащий целое число
>>> y
(40,)
```

В особом случае Python позволяет не указывать открывающую и закрывающую круглые скобки для кортежа в контекстах, где это синтаксически однозначно. Например, в четвертой строке табл. 9.1 просто перечислены четыре элемента, разделенные запятыми. В контексте оператора присваивания Python распознает такое как кортеж, несмотря на отсутствие круглых скобок.

Одни будут говорить вам о необходимости всегда использовать в кортежах круглые скобки, а другие – никогда не применять их в кортежах (найдутся и те, кто вообще ничего не скажет о том, что делать с вашими кортежами!). Наиболее распространенными местами, в которых скобки обязательны для литералов кортежей, являются те, где:

- круглые скобки имеют значение – кортеж находится внутри вызова функции или вложен в более крупное выражение;
- запятые имеют значение – кортеж встроен в литерал более крупной структуры данных вроде списка или словаря либо перечислен в операторе `print` версии Python 2.X.

В большинстве других контекстов заключающие круглые скобки необязательны. Хороший совет начинающим: вероятно легче использовать круглые скобки, чем помнить о том, когда они необязательны, а когда обязательны. Многие программисты также считают, что круглые скобки способствуют лучшей читабельности сценариев, делая кортежи более явными и очевидными¹.

Преобразования, методы и неизменяемость

За исключением различий в синтаксисе литералов операции над кортежами (строки в середине табл. 9.1) идентичны строковым и списковым операциям. Единственное отличие, заслуживающее внимания, связано с тем, что операции `+`, `*` и нарезания возвращают новые *кортежи*, когда применяются к кортежам, и кортежи не предоставляют те же методы, которые вы видели для строк, списков и словарей. Скажем, если нужно отсортировать кортеж, то обычно придется либо сначала преобразовать его в список, чтобы получить доступ к методу сортировки и сделать его изменяемым объектом, либо использовать более новую встроенную функцию `sorted`, принимающую любой объект последовательности (и другие *имеющиеся* объекты – термин, введенный в главе 4, который будет более формально определен в следующей части книги):

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)                  # Создание списка из элементов кортежа
>>> tmp.sort()                   # Сортировка списка
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)                # Создание кортежа из элементов списка
>>> T
('aa', 'bb', 'cc', 'dd')
```

¹ Более тонкий фактор: запятая представляет собой своего рода операцию с самым низким приоритетом, но только в контекстах, где она иначе не является значащей. В таких ситуациях кортежи формируются именно запятой, а не круглыми скобками; это делает круглые скобки необязательными, но также приводит к странным и неожиданным синтаксическим ошибкам, если они опущены.

```
>>> sorted(T)          # Либо использование встроенной функции sorted и
экономия двух шагов
['aa', 'bb', 'cc', 'dd']
```

Встроенные функции `list` и `tuple` применяются для преобразования объекта в список и затем обратно в кортеж; в действительности оба вызова создают новые объекты, но совокупный эффект похож на преобразование.

Преобразовывать кортежи можно также с использованием списковых включений. Следующие команды создают список из кортежа, попутно добавляя 20 к каждому элементу:

```
>>> T = (1, 2, 3, 4, 5)
>>> L = [x + 20 for x in T]
>>> L
[21, 22, 23, 24, 25]
```

Списковые включения на самом деле представляют собой операции над последовательностями — они всегда строят новые списки, но могут применяться для прохода по любым объектам последовательностей, включая кортежи, строки и другие списки. Как мы увидим позже в книге, они даже работают с тем, что не является физически хранящимися последовательностями — подойдут любые *итерируемые* объекты, в том числе файлы, которые автоматически читаются строка за строкой. Учитывая такой факт, их лучше стоило бы называть *итерационными* инструментами.

Хотя кортежи не располагают такими же методами, как списки и строки, начиная с версий Python 2.6 и 3.0, они имеют два собственных метода — `index` и `count` работают так же, как для списков, но определены для объектов кортежей:

```
>>> T = (1, 2, 3, 2, 4, 2)      # Методы кортежей в Python 2.6, 3.0
                                # и последующих версиях
>>> T.index(2)                # Смещение первого появления элемента 2
1
>>> T.index(2, 2)              # Смещение появления элемента 2 после смещения 2
3
>>> T.count(2)                # Сколько всего элементов 2?
3
```

До выхода Python 2.6 и 3.0 у кортежей вообще отсутствовали методы — это старое соглашение Python для неизменяемых типов, которое по соображениям практичности было нарушено несколько лет назад для строк и недавно для чисел и кортежей.

Кроме того, обратите внимание, что правило о неизменяемости кортежей применяется только к верхнему уровню самого кортежа, но не к его содержимому. Например, список внутри кортежа можно изменять обычным образом:

```
>>> T = (1, [2, 3], 4)
>>> T[1] = 'spam'               # Потерпит неудачу: сам кортеж изменять нельзя
TypeError: object doesn't support item assignment
Ошибка типа: объект не поддерживает присваивание в отношении элементов
>>> T[1][0] = 'spam'            # Работает: изменяемые объекты внутри кортежа
                                # можно модифицировать
>>> T
(1, ['spam', 3], 4)
```

В большинстве программ такой одноуровневой неизменяемости вполне достаточно для распространенных сценариев использования кортежей, которые обсуждаются в следующем разделе.

Для чего используются списки и кортежи?

Похоже, что при изучении кортежей новичками всегда возникает вопрос: зачем нам применять кортежи, если у нас есть списки? Часть аргументации связана с историей; создатель Python по образованию математик, и ему приписывают видение кортежа как простого объединения объектов, а списка — как структуры данных, которая изменяется со временем. На самом деле такое использование слова “кортеж” происходит из математики, а также объясняется частым применением для строк таблиц в реляционных базах данных.

Однако наилучший ответ, по-видимому, заключается в том, что неизменяемость кортежей обеспечивает определенную степень *целостности* — вы можете быть уверены, что кортеж не изменится через ссылку где-то в другом месте программы, но для списков такая гарантия отсутствует. Следовательно, кортежи и остальные неизменяемые объекты исполняют роль, похожую на объявления “констант” в других языках, хотя понятие константности в Python ассоциировано с объектами, а не с переменными.

Кроме того, кортежи могут использоваться в местах, где нельзя применять списки — скажем, как словарные ключи (см. пример разреженной матрицы в главе 8). Некоторые встроенные операции также могут требовать или подразумевать указание кортежей вместо списков (например, значения подстановки в выражении форматирования строки), хотя в последние годы такие операции были обобщены, чтобы сделать их более гибкими. Запомните в качестве эмпирического правила, что списки являются предпочтительным инструментом для упорядоченных коллекций, которые могут нуждаться в изменении, а кортежи способны справиться с другими сценариями фиксированных объединений.

Снова о записях: именованные кортежи

В действительности выбор типов данных оказывается даже богаче, чем могло вытекать из предыдущего раздела — нынешним программистам на Python доступен широкий ассортимент встроенных основных типов и типов расширений, построенных на их основе. Скажем, во врезке “Что потребует внимания: словари или списки” в предыдущей главе было показано, как представить информацию, подобную записи, с помощью списка и словаря, а также отмечено, что словари обладают преимуществом более осмысленных ключей по сравнению с помеченными данными. До тех пор пока нам не требуется изменяемость, кортежи могут выполнять похожие роли с позициями для полей записи подобно спискам:

```
>>> bob = ('Bob', 40.5, ['dev', 'mgr'])      # Запись в виде кортежа
>>> bob
('Bob', 40.5, ['dev', 'mgr'])
>>> bob[0], bob[2]                          # Доступ по позиции
('Bob', ['dev', 'mgr'])
```

Тем не менее, как и в списках, номера полей в целом несут меньше информации, чем имена ключей в *словаре*. Вот та же запись, представленная в виде словаря с именованными полями:

```
>>> bob = dict(name='Bob', age=40.5, jobs=['dev', 'mgr']) # Запись в виде словаря
>>> bob
{'jobs': ['dev', 'mgr'], 'name': 'Bob', 'age': 40.5}
>>> bob['name'], bob['jobs']                      # Доступ по ключу
('Bob', ['dev', 'mgr'])
```

На самом деле при необходимости мы можем преобразовывать части словаря в кортеж:

```
>>> tuple(bob.values())          # Преобразование значений в кортеж
(['dev', 'mgr'], 'Bob', 40.5)
>>> list(bob.items())           # Преобразование элементов в список кортежей
[('jobs', ['dev', 'mgr']), ('name', 'Bob'), ('age', 40.5)]
```

Но проделав небольшую дополнительную работу, мы можем реализовать объекты, которые предлагают доступ к полям записи и по позиции, и по имени. Например, в стандартном библиотечном модуле `collections`, упомянутом в главе 8, имеется класс `namedtuple`. Он реализует тип расширения, добавляя к кортежам логику, которая делает возможным доступ к компонентам по *позиции* и по *имени* атрибута, и при желании может быть преобразован в форму словарного вида для доступа по *ключу*. Имена атрибутов происходят из классов и не являются в точности словарными ключами, но они в аналогичной степени осмысленны:

```
>>> from collections import namedtuple      # Импортирование типа расширения
>>> Rec = namedtuple('Rec', ['name', 'age', 'jobs']) # Создание
                                                # производного класса
>>> bob = Rec('Bob', age=40.5, jobs=['dev', 'mgr']) # Запись в виде
                                                # именованного кортежа
>>> bob
Rec(name='Bob', age=40.5, jobs=['dev', 'mgr'])
>>> bob[0], bob[2]                           # Доступ по позиции
('Bob', ['dev', 'mgr'])
>>> bob.name, bob.jobs                      # Доступ по атрибуту
('Bob', ['dev', 'mgr'])
```

В случае необходимости именованный кортеж можно преобразовать в словарь, чтобы обеспечить поведение, основанное на ключах:

```
>>> O = bob._asdict()          # Форма, похожая на словарь
>>> O['name'], O['jobs']       # Доступ также и по ключу
('Bob', ['dev', 'mgr'])
>>> O
OrderedDict([('name', 'Bob'), ('age', 40.5), ('jobs', ['dev', 'mgr'])])
```

Как видите, именованные кортежи представляют собой *гибрид* кортежа/класса/словаря. Они также демонстрируют классический *компромисс*. В обмен на дополнительную практичность они требуют написания добавочного кода (две строки в начале предыдущего примера, которые импортируют тип и создают класс), а также влекут за собой определенные затраты в плане производительности. (Короче говоря, именованные кортежи строят новые классы, расширяющие тип кортежей, за счет вставки для каждого именованного поля метода доступа `property`, который отображает имя на его позицию. Прием опирается на более сложные темы, исследуемые в части VIII, и использует форматированные строки кода вместо инструментов аннотирования классов, таких как декораторы и метаклассы.) И все же они являются хорошим примером разновидности специальных типов данных, которые мы можем создавать на основе встроенных типов вроде кортежей, когда желательно иметь дополнительную полезность.

Именованные кортежи доступны в Python 3.X, 2.7, 2.6 (где `_asdict` возвращает подлинный словарь) и возможно более ранних версиях, хотя они полагаются на относительно новые средства в стандартах Python. Они также представляют собой *расширения*, а не основные типы (находятся в стандартной библиотеке и относятся к той

же категории, что и типы `Fraction` и `Decimal` из главы 5), поэтому ищите сведения о них в руководстве по библиотеке Python.

Однако кратко отметим, что кортежи и именованные кортежи поддерживают распаковывающее присваивание кортежей, которое мы формально обсудим в главе 13, а также итерационные контексты, исследуемые в главах 14 и 20 (обратите внимание на позиционные начальные значения: именованные кортежи принимают их по имени, по позиции или по тому и другому):

```
>>> bob = Rec('Bob', 40.5, ['dev', 'mgr']) # Для кортежей и именованных кортежей
>>> name, age, jobs = bob                  # Присваивание кортежей (глава 11)
>>> name, jobs
('Bob', ['dev', 'mgr'])

>>> for x in bob: print(x)                 # Итерационный контекст (главы 14, 20)
...выводит Bob, 40.5, ['dev', 'mgr']...
```

Распаковывающее присваивание кортежей не вполне применимо к словарям, если не считать извлечения и преобразования ключей и значений, а также допущения или навязывания им позиционного упорядочения (словари не являются последовательностями), и итерация проходит по ключам, а не по значениям (обратите внимание на форму словарного литерала как альтернативу `dict`):

```
>>> bob = {'name': 'Bob', 'age': 40.5, 'jobs': ['dev', 'mgr']}
>>> job, name, age = bob.values()
>>> name, job          # Эквивалент dict (но порядок может варьироваться)
('Bob', ['dev', 'mgr'])

>>> for x in bob: print(bob[x])    # Проход по ключам, индексация значений
...выводит значения...
```

Дождитесь финальной переделки этого представления записи в главе 27, где будет показано, каким образом в общую картину вписываются *классы*, определяемые пользователем; как выяснится, классы также обозначают поля именами, но могут предоставить еще и программную логику для обработки данных записи в той же самой упаковке.

Файлы

Возможно, вы уже знакомы с понятием файлов, которые представляют собой именованные ячейки хранилища на компьютере, управляемые операционной системой. Последний исследуемый крупный встроенный тип основных объектов обеспечивает способ доступа к таким файлам внутри программ Python.

Вкратце встроенная функция `open` создает файловый объект Python, который служит ссылкой на файл, находящийся на вашем компьютере. После вызова `open` вы можете передавать строки данных в и из ассоциированного внешнего файла, вызывая методы возвращенного файлового объекта.

По сравнению с типами, которые вы видели до сих пор, файловые объекты кое в чем необычны. Они считаются основным типом, потому что создаются встроенной функцией, но отличаются от чисел, последовательностей или отображений, а также не реагируют на операции выражений; файловые объекты экспортируют только методы для решения распространенных задач обработки файлов. Большинство файловых методов занимаются выполнением ввода из и вывода во внешний файл, ассоциированный с файловым объектом, но остальные файловые методы позволяют переходить в новую позицию внутри файла, сбрасывать буферы вывода и т.д. В табл. 9.2 приведен обзор часто применяемых файловых операций.

Таблица 9.2. Распространенные файловые операции

Операция	Описание
<code>output = open(r'C:\spam', 'w')</code>	Создает выходной файл ('w' означает write — запись)
<code>input = open('data', 'r')</code>	Создает входной файл ('r' означает read — чтение)
<code>input = open('data')</code>	То же, что и в предыдущей строке ('r' выбирается по умолчанию)
<code>aString = input.read()</code>	Читает целый файл в одиночную строку
<code>aString = input.read(N)</code>	Читает до N следующих символов (или байтов) в строку
<code>aString = input.readline()</code>	Читает следующую строку файла (включая символ новой строки \n) в строку
<code>aList = input.readlines()</code>	Читает целый файл в список строк (с символами \n)
<code>output.write(aString)</code>	Записывает строку символов (или байтов) в файл
<code>output.writelines(aList)</code>	Записывает все строки из списка в файл
<code>output.close()</code>	Вручную закрывает файл (это делается автоматически, когда файловый объект подвергается сборке мусора)
<code>output.flush()</code>	Сбрасывает буфер вывода на диск, не закрывая файл
<code>anyFile.seek(N)</code>	Изменяет позицию на N для следующей операции
<code>for line in open('data'): использовать строку</code>	Файловые итераторы, читающие строку за строкой
<code>open('f.txt', encoding='latin-1')</code>	Текстовые файлы Unicode в Python 3.X (строки str)
<code>open('f.bin', 'rb')</code>	Байтовые файлы в Python 3.X (строки bytes)
<code>codecs.open('f.txt', encoding='utf8')</code>	Текстовые файлы Unicode в Python 2.X (строки unicode)
<code>open('f.bin', 'rb')</code>	Байтовые файлы в Python 2.X (строки str)

Открытие файлов

Для открытия файла в программе вызывается встроенная функция `open` с указанием имени внешнего файла и режима обработки. Вызов возвращает файловый объект, имеющий методы для передачи данных:

```
afile = open(имя_файла, режим)
afile.метод()
```

Первый аргумент функции `open`, имя внешнего файла, может включать префикс со специфичным для платформы и абсолютным или относительным путем к каталогу. Без пути к каталогу файл считается находящимся в текущем рабочем каталоге (т.е. там, где запускается сценарий). Как вы увидите в расширенном описании файлов в главе 37, имя файла может также содержать отличные от ASCII символы Unicode, которые Python автоматически транслирует в и из внутренней кодировки платформы, или предоставляться в виде предварительно закодированной байтовой строки.

Во втором аргументе функции `open` (режим обработки) обычно указывается строка '`r`', чтобы открыть файл для текстового ввода (по умолчанию), '`w`' — чтобы создать и открыть файл для текстового вывода, либо '`a`' — чтобы открыть файл для дополнения текстом в конце (например, для добавления в журнальные файлы). Аргумент режима обработки может задавать дополнительные параметры:

- добавление `b` в строку режима разрешает работу с двоичными данными (трансляции символов конца строки и кодировки Unicode в Python 3.X отключаются);
- добавление `+` в строку режима приводит к открытию файла для ввода и вывода (т.е. вы можете выполнять чтение и запись в один и тот же файловый объект часто в сочетании с операциями позиционирования в файле).

Первые два аргумента функции `open` должны быть строками Python. Необязательный третий аргумент используется для управления *буферизацией* вывода. Указание в нем нуля означает, что вывод не буферизируется (он передается внешнему файлу немедленно при вызове метода записи), а для файлов специальных типов могут быть предоставлены дополнительные аргументы (скажем, *кодировка* для текстовых файлов Unicode в Python 3.X).

Здесь мы раскроем фундаментальные основы файлов и рассмотрим ряд базовых примеров, но не будем погружаться во все параметры режимов обработки файлов; как всегда, исчерпывающие сведения ищите в руководстве по библиотеке Python.

Использование файлов

После создания файлового объекта посредством `open` можно вызывать его методы для чтения и записи в ассоциированный внешний файл. Во всех случаях текст из файла принимает форму строк в программах Python; методы чтения файла возвращают его содержимое в строках и содержимое передается методам записи в виде строк. Методы чтения и записи имеют множество вариантов, наиболее распространенные из которых перечислены в табл. 9.2. Ниже приведено несколько замечаний относительно применения файлов.

Файловые итераторы лучше всего подходят для чтения строк

Несмотря на обилие в табл. 9.2 методов чтения и записи, имейте в виду, что вероятно наилучший на сегодняшний день способ чтения строк из текстового файла — не читать файл вообще. Как вы увидите в главе 14, файлы также имеют *итератор*, который автоматически читает по одной строке за раз в цикле `for`, списковом включении или другом итерационном контексте.

Содержимое является строками, а не объектами

Обратите внимание в табл. 9.2, что данные, *прочитанные* из файла, всегда возвращаются в сценарий в виде строки, поэтому ее придется преобразовывать в объект Python другого типа, если строка — не то, что нужно. Аналогично в отличие от операции `print` при записи данных в файл Python не добавляет форматирование и не преобразует объекты в строки автоматически — вы обязаны

предоставлять уже сформатированные строки. По этой причине при работе с файлами будут полезными инструменты, предназначенные для преобразования объектов в и из строк (например, `int`, `float`, `str` и выражение и метод формирования строк).

В состав Python также входят стандартные библиотечные инструменты для поддержки обобщенного хранилища объектов (модуль `pickle`), для работы с упакованными двоичными данными в файлах (модуль `struct`) и для обработки специальных типов содержимого, таких как текст JSON, XML и CSV. Они будут демонстрироваться позже в главе и книге, а полное их описание доступно в руководствах по Python.

Файлы буферизуются и поддерживаются позиционирование

По умолчанию выходные файлы всегда *буферизуются*, а это значит, что записываемый текст может не сразу быть передан из памяти на диск — сбрасывание буферизированных данных на диск инициирует закрытие файла или вызов метода `flush`. Избежать буферизации позволяют дополнительные аргументы функции `open`, но тогда может пострадать производительность. Файлы Python также поддерживают *произвольный доступ* на основе байтовых смещений — их метод `seek` дает возможность переходить в специфические позиции для чтения и записи.

Вызов метода `close` часто необязателен: автоматическое закрытие при сборке мусора

Вызов метода `close` прекращает связь с внешним файлом, освобождает системные ресурсы и сбрасывает буферизированный вывод на диск, если он все еще находится в памяти. Как обсуждалось в главе 6, область памяти объекта в Python автоматически освобождается, как только на объект перестают ссылаться где-нибудь в программе. Когда освобождается память, занимаемая *файловыми* объектами, Python также автоматически *закрывает* файлы, если они по-прежнему открыты (это также происходит при завершении работы программы).

Таким образом, в стандартном Python закрывать файлы вручную не всегда обязательно, особенно файлы в простых сценариях с коротким временем выполнения и временные файлы, используемые в одной строке кода или выражении.

С другой стороны, включение вызовов `close` ничем не вредит и может стать хорошей привычкой, особенно в длительно функционирующих системах. Строго говоря, средство автоматического закрытия при сборке мусора не является частью определения языка. Со временем оно может измениться, происходить не тогда, когда вы ожидаете его в интерактивной подсказке, и не работать одинаково в других реализациях Python, в которых сборщики мусора могут освобождать память и закрывать файлы не в те же самые моменты, что и стандартный CPython. На самом деле, когда множество файлов открывается внутри циклов, реализации Python, отличающиеся от CPython, могут требовать вызова `close` для немедленного освобождения системных ресурсов, прежде чем сборщик мусора займется освобождением памяти, занимаемой объектами. Кроме того, вызовы `close` иногда могут требоваться для сбрасывания буферизированного вывода файловых объектов, память которых пока еще не освобождена. Альтернативный способ гарантирования автоматического закрытия файлов описан позже в текущем разделе при обсуждении *диспетчера контекста* файлового объекта, применяемого с оператором `with/as` в Python 2.6, 2.7 и 3.X.

Файлы в действии

Давайте рассмотрим простой пример, который продемонстрирует основы обработки файлов. В следующем коде открывается новый текстовый файл для вывода, в него записываются две строки (завершающиеся символом новой строки \n) и файл закрывается. Позже тот же файл снова открывается, и из него читаются строки по одной за раз с помощью метода readline. Обратите внимание, что третий вызов readline возвращает пустую строку; именно так файловые методы Python сообщают о том, что достигнут конец файла (пустые строки в файле возвращаются как строки, содержащие только символ новой строки, а не по-настоящему пустые строки). Ниже приведено полное взаимодействие:

```
>>> myfile = open('myfile.txt', 'w')      # Открытие файла для текстового
                                             # вывода: создание/очистка
>>> myfile.write('hello text file\n') # Запись строки текста: строковый объект
16
>>> myfile.write('goodbye text file\n')
18
>>> myfile.close()                      # Сброс выходных буферов на диск
>>> myfile = open('myfile.txt')          # Открытие файла для текстового ввода:
                                             # 'r' принимается по умолчанию
>>> myfile.readline()                   # Чтение строк
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline()                  # Пустая строка: конец файла
''
```

Вызовы файлового метода write возвращают количество записанных символов в Python 3.X, но не в Python 2.X, так что выводиться интерактивно они не будут. Каждая строка текста, включая маркер конца строки \n, в примере записывается как строковый объект; методы записи не добавляют символ конца строки, поэтому его понадобится добавить, чтобы надлежащим образом завершить строки текста (иначе следующая запись просто расширят текущую строку в файле).

Если вы хотите отобразить содержимое файла с интерпретацией символов конца строки, тогда прочитайте *сразу* весь файл в строковый объект посредством файлового метода read и выведите его:

```
>>> open('myfile.txt').read() # Чтение сразу всего файла в строковый объект
'hello text file\ngoodbye text file\n'
>>> print(open('myfile.txt').read()) # Дружественное к пользователю отображение
hello text file
goodbye text file
```

При желании файл можно просматривать построчно с помощью *файловых итераторов*:

```
>>> for line in open('myfile.txt'):      # Использование файловых итераторов,
                                             # а не чтения
...     print(line, end='')
...
hello text file
goodbye text file
```

В данном случае временный файловый объект, созданный функцией open, будет автоматически читать и возвращать одну строку на каждой итерации цикла. Такая форма обычно легче для кодирования, эффективнее расходует память и может оказаться

быстрее ряда других вариантов (конечно, в зависимости от многих условий). Тем не менее, мы еще не добрались до операторов и итераторов, поэтому с более полным объяснением приведенного кода придется повременить до главы 14.



Примечание для пользователей Windows. Как упоминалось в главе 7, функция `open` принимает прямые косые черты в стиле Unix вместо обратных косых черт, обычных для Windows, поэтому для путей к каталогам подойдет любая из указанных ниже форм – неформатированные строки, прямые косые черты или удвоенные обратные косые черты:

```
>>> open(r'C:\Python33\Lib\pdb.py').readline()
'#! /usr/bin/env python3\n'
>>> open('C:/Python33/Lib/pdb.py').readline()
'#! /usr/bin/env python3\n'
>>> open('C:\\Python33\\Lib\\pdb.py').readline()
'#! /usr/bin/env python3\n'
```

Форма с неформатированными строками в первой команде полезна еще и для отключения случайных управляемых последовательностей, когда контролировать содержимое строк невозможно, а также при других обстоятельствах.

Кратко о текстовых и двоичных файлах

Строго говоря, в примере из предыдущего раздела использовался текстовый файл. В Python 3.X и 2.X тип файла определяется вторым аргументом функции `open`, строкой режима – наличие в ней `b` означает двоичный файл. В Python всегда поддерживались как текстовые, так и двоичные файлы, но в Python 3.X между ними имеется более строгое разграничение.

- Текстовые файлы представляют содержимое как нормальные строки `str`, автоматически выполняют кодирование и декодирование Unicode и по умолчанию производят трансляцию символов конца строки.
- Двоичные файлы представляют содержимое как специальный строковый тип `bytes` и позволяют программам получать доступ к неизмененному содержимому файлов.

Напротив, текстовые файлы Python 2.X поддерживают 8-битный текст и двоичные данные, а текст Unicode обрабатывается специальным строковым типом и файловым интерфейсом (строки `unicode` и `codecs.open`). Отличия в Python 3.X произрастают из того факта, что в нормальном строковом типе простой текст и текст Unicode были объединены – это имеет смысл, т.к. весь текст представлен в Unicode, включая ASCII и другие 8-битные кодировки.

Поскольку большинство программистов имеют дело только с текстом ASCII, они могут обойтись базовым интерфейсом к текстовым файлам, который применялся в предыдущем примере, и нормальными строками. В Python 3.X все строки формально представлены в Unicode, но пользователи ASCII обычно этого не замечают. Фактически текстовые файлы и строки работают одинаково в Python 3.X и 2.X, если охват сценария ограничен такими простыми формами текста.

Однако если вам нужно поддерживать интернационализированные приложения или байтовые данные, тогда различие в Python 3.X повлияет на ваш код (обычно в лучшую сторону). В общем случае вы должны использовать строки `bytes` для двоичных файлов и нормальные строки `str` для текстовых файлов. Вдобавок из-за того, что тек-

стовые файлы реализуют кодировки Unicode, вы не должны открывать файл двоичных данных в текстовом режиме — декодирование его содержимого в текст Unicode, скорее всего, потерпит неудачу.

Рассмотрим пример. Когда вы читаете файл *двоичных* данных, то получаете обратно объект `bytes` — последовательность небольших целых чисел, представляющих абсолютные байтовые значения (соответствующие или не соответствующие символам), которая выглядит почти как нормальная строка. Вот код для Python 3.X, исходя из предположения, что двоичный файл существует:

```
>>> data = open('data.bin', 'rb').read() # Открытие двоичного файла:  
#     rb = чтение двоичный  
>>> data  
b'\x00\x00\x00\x07spam\x00\x08'  
>>> data[4:8] # Действует подобно строкам  
b'spam'  
>>> data[4:8][0] # Но на самом деле является 8-битными  
#     целыми числами  
115  
>>> bin(data[4:8][0]) # Функция bin() из Python 3.X/2.6+  
'0b1110011'
```

Кроме того, двоичные файлы не производят *трансляцию символов конца строки* в данных; *текстовые* файлы по умолчанию отображают все их формы в `\n` и обратно во время записи и чтения, а также реализуют кодировки Unicode при передачах в Python 3.X. Двоичные файлы вроде показанного выше работают в Python 2.X точно так же, но байтовые строки представляют собой просто нормальные строки и при выводе не имеют ведущих символов `b`, а текстовые файлы для добавления обработки Unicode обязаны применять модуль `codecs`.

Тем не менее, согласно примечанию в начале главы, это все, что мы собирались здесь обсудить о тексте Unicode и файлах двоичных данных, и его вполне достаточно для понимания предстоящих примеров в главе. Так как различие представляет лишь незначительный интерес для многих программистов на Python, мы предлагаем обзор файлов в главе 4, а более подробные исследования — в главе 37. А теперь перейдем к более существенным примерам работы с файлами, чтобы продемонстрировать несколько распространенных сценариев использования.

Хранение объектов Python в файлах: преобразования

В следующем примере производится запись разнообразных объектов Python в текстовый файл во множество строк. Обратите внимание, что объекты должны быть преобразованы в строки с применением инструментов преобразования. Опять-таки данные файла всегда являются строками в наших сценариях, и методы записи не делают никакого автоматического форматирования строк (ради экономии места счетчики байтов, выводимые вызовами `write`, здесь не показаны):

```
>>> X, Y, Z = 43, 44, 45      # Собственные объекты Python  
>>> S = 'Spam'                # Для сохранения в файле обязаны быть строками  
>>> D = {'a': 1, 'b': 2}  
>>> L = [1, 2, 3]  
>>>  
>>> F = open('datafile.txt', 'w') # Создание выходного текстового файла  
>>> F.write(S + '\n')          # Завершение строк символом \n  
>>> F.write('%s,%s,%s\n' % (X, Y, Z)) # Преобразование чисел в строки
```

```
>>> F.write(str(L) + '$' + str(D) + '\n')      # Преобразование и разделение
                                         # посредством $
>>> F.close()
```

После того, как файл создан, мы можем просмотреть его содержимое, открыв и прочитав в строковый объект (объединенные здесь в единственную операцию). Интерактивный вывод дает точное содержимое байтов, а операция `print` интерпретирует встроенные символы конца строки для визуализации более дружественного к пользователю представления:

```
>>> chars = open('datafile.txt').read()    # Отображение неформатированной строки
>>> chars
"Spam\n43,44,45\n[1, 2, 3]$('a': 1, 'b': 2)\n"
>>> print(chars)                         # Дружественное к пользователю отображение
Spam
43,44,45
[1, 2, 3]$('a': 1, 'b': 2)
```

Теперь нам нужно использовать другие инструменты преобразования для трансляции строк из текстового файла в действительные объекты Python. Так как Python никогда не преобразовывает строки в числа (или в объекты других типов) автоматически, это обязательно, если необходимо иметь доступ к обычным возможностям объектов, подобным индексации, сложения и т.д.:

```
>>> F = open('datafile.txt')           # Открытие файла
>>> line = F.readline()             # Чтение одной строки
>>> line
'Spam\n'
>>> line.rstrip()                  # Удаление символа конца строки
'Spam'
```

Строковый метод `rstrip` применяется, чтобы избавиться от хвостового символа конца строки; срез `line[:-1]` также работал бы, но только когда мы уверены в том, что все строки заканчиваются символом `\n` (иногда он может отсутствовать в последней строке файла).

До сих пор мы прочитали из файла строку, содержащую последовательность символов. Теперь давайте захватим следующую строку, которая содержит числа, и разберем (т.е. извлечем) объекты из нее:

```
>>> line = F.readline()           # Чтение следующей строки из файла
>>> line                         # Строковый объект
'43,44,45\n'
>>> parts = line.split(',')      # Разбиение (разбор) по запятым
>>> parts
['43', '44', '45\n']
```

Мы используем строковый метод `split` для разбиения строки по разделителям-запятым; результатом будет список подстрок с индивидуальными числами. Однако если мы хотим выполнять с ними математические действия, то нам еще придется преобразовать подстроки в целые числа:

```
>>> int(parts[1])                # Преобразование строки в целое число
44
>>> numbers = [int(P) for P in parts] # Преобразование всего списка за раз
>>> numbers
[43, 44, 45]
```

Как вы знаете, функция `int` транслирует строку цифр в объект целого числа, а выражение спискового включения, представленное в главе 4, может применить ее к каждому элементу в списке за раз (списковые включения рассматриваются позже в книге). Обратите внимание, что мы не обязаны выполнять `rstrip` для удаления символа `\n` в конце последней части; `int` и ряд других преобразователей молча игнорируют пробельные символы вокруг цифр.

Наконец, чтобы преобразовать список и словарь, сохраненные в третьей строке файла, мы можем прогнать их через `eval` – встроенную функцию, которая трактует строку как порцию исполняемого программного кода (формально как строку, содержащую выражение Python):

```
>>> line = F.readline()
>>> line
"[1, 2, 3]$('a': 1, 'b': 2)\n"
>>> parts = line.split('$')           # Разбиение (разбор) по $
>>> parts
[['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]
>>> eval(parts[0])                  # Преобразование в объект любого типа
[1, 2, 3]
>>> objects = [eval(P) for P in parts] # То же самое для всего списка
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```

Поскольку конечным результатом всего разбора и преобразования является список нормальных объектов Python, а не строк, теперь мы можем применять к ним списковые и словарные операции в своем сценарии.

Хранение собственных объектов Python: модуль `pickle`

Использование функции `eval` для преобразования строк в объекты, как демонстрировалось выше в коде, представляет собой мощный прием. Фактически временами он *чрезвычайно* мощный. Функция `eval` благополучно выполнит любое выражение Python – даже то, которое способно удалить все файлы на вашем компьютере, располагая необходимыми полномочиями! Если вы действительно хотите сохранять собственные объекты Python, но не можете доверять источнику данных в файле, тогда идеально подойдет стандартный библиотечный модуль Python по имени `pickle`.

Модуль `pickle` является более развитым инструментом, который позволяет сохранять почти любой объект Python в файл напрямую, не требуя с нашей стороны каких-либо преобразований в и из строки. Он похож на крайне универсальную утилиту форматирования и разбора данных. Скажем, вот как сохранить словарь в файл:

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)      # Сохранение любого объекта в файл с помощью pickle
>>> F.close()
```

Чтобы позже получить словарь обратно, мы просто снова применяем `pickle` для его воссоздания:

```
>>> F = open('datafile.pkl', 'rb')
>>> E = pickle.load(F)      # Загрузка любого объекта из файла
>>> E
{'a': 1, 'b': 2}
```

Мы получаем эквивалентный объект словаря, не предпринимая никаких разбиений или преобразований вручную. Модуль `pickle` выполняет то, что известно как *сериализация объектов* (преобразование объектов в строки байтов и обратно), но требует совсем небольшой работы с нашей стороны. На самом деле модуль `pickle` внутренне транслирует словарь в строковую форму, которая не выглядит особо привлекательно (к тому же может варьироваться при использовании `pickle` в других режимах протокола данных):

```
>>> open('datafile.pkl', 'rb').read() # Формат предрасположен к изменениям!
b'\x80\x03q\x00\x01\x00\x00bq\x01K\x02X\x01\x00\x00\x00aq\x02K\x01u.'
```

Поскольку модуль `pickle` в состоянии реконструировать объект из такого формата, нам не придется делать это самостоятельно. Дополнительные сведения о модуле `pickle` ищите в руководстве по стандартной библиотеке Python либо импортируйте `pickle` и передайте его `help` в интерактивной подсказке. Во время исследований взгляните также на модуль `shelve`, который представляет собой инструмент, применяющий `pickle` для хранения объектов Python в файловой системе с доступом по ключу, что выходит за рамки рассматриваемых здесь тем (хотя вы увидите пример `shelve` в действии в главе 28, а также другие примеры использования `pickle` в главах 31 и 37 тома 2).



Обратите внимание, что файл, который применяется для хранения объекта, обработанного `pickle`, был открыт в *двоичном режиме*; в Python 3.X двоичный режим всегда обязателен, т.к. `pickle` создает и использует строковый объект `bytes`, а такие объекты подразумевают файлы в двоичном режиме (файлы в текстовом режиме в Python 3.X подразумеваются строки `str`). В ранних версиях Python допускалось применять файлы в текстовом режиме для протокола 0 (принимается по умолчанию и создает текст ASCII), пока текстовый режим используется согласованно; более высокие протоколы требуют файлов в двоичном режиме. Стандартным протоколом Python 3.X является 3 (двоичный), но он создает строковый объект `bytes` даже для протокола 0. Дополнительную информацию и примеры применения модуля `pickle` ищите в главах 28, 31 и 37 второго тома, руководстве по библиотеке Python или справочниках.

В Python 2.X также имеется модуль `cPickle`, который представляет собой оптимизированную версию `pickle` и может импортироваться напрямую для достижения более высокой скорости. В Python 3.X этот модуль переименован в `_pickle` и используется автоматически в `pickle` – сценарии просто импортируют `pickle` и позволяют Python самому заняться оптимизацией.

Хранение объектов Python в формате JSON

Модуль `pickle` из предыдущего раздела транслирует произвольные объекты Python в патентованный формат, разработанный специально для Python и на протяжении многих лет отрегулированный для достижения высокой производительности. JSON является более новым и развивающимся форматом обмена данными, нейтральным к языкам программирования и поддерживаемым различными системами. Например, *MongoDB* хранит данные в документной базе данных, в которой применяется двоичный формат JSON.

Формат JSON не поддерживает настолько широкий диапазон типов объектов Python, как модуль `pickle`, но его переносимость становится преимуществом в ряде контекстов, и он представляет еще один способ сериализации специфической категории объектов Python для хранения и передачи. Кроме того, из-за синтаксической

близости JSON к словарям и спискам Python трансляция объектов Python в него и обратно тривиальна и автоматизируется стандартным библиотечным модулем `json`.

Скажем, словарь Python с вложенными структурами очень похож на данные JSON, хотя переменные и выражения Python поддерживают более богатые варианты структурирования (и часть показанного далее может быть произвольным выражением в коде Python):

```
>>> name = dict(first='Bob', last='Smith')
>>> rec = dict(name=name, job=['dev', 'mgr'], age=40.5)
>>> rec
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
```

Отображаемый здесь финальный словарный формат является допустимым литералом в коде Python и передается для JSON почти в том виде, как есть, но модуль `json` делает трансляцию официальной – ниже демонстрируется трансляция объектов Python в сериализованное строковое представление JSON в памяти и обратно:

```
>>> import json
>>> json.dumps(rec)
'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "age": 40.5}'
>>> S = json.dumps(rec)
>>> S
'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "age": 40.5}'
>>> O = json.loads(S)
>>> O
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
>>> O == rec
True
```

Подобным же образом просто транслировать объекты Python в и из строк данных JSON в файлах. До сохранения в файле ваши данные существуют как объекты Python; модуль JSON воссоздает их из текстового представления JSON, когда загружает его из файла:

```
>>> json.dump(rec, fp=open('testjson.txt', 'w'), indent=4)
>>> print(open('testjson.txt').read())
{
    "job": [
        "dev",
        "mgr"
    ],
    "name": {
        "last": "Smith",
        "first": "Bob"
    },
    "age": 40.5
}
>>> P = json.load(open('testjson.txt'))
>>> P
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
```

После трансляции из текста JSON вы обрабатываете данные с использованием привычных операций над объектами Python в своем сценарии. Дополнительные сведения о JSON ищите в руководстве по библиотеке Python и в веб-сети.

Обратите внимание, что все строки в JSON представлены в *Unicode*, чтобы поддерживать текст с интернациональными таблицами символов, и потому после трансля-

ции из JSON вы увидите в строках ведущие символы и в Python 2.X (но не в Python 3.X); как кратко упоминалось в главах 4 и 7 и полностью раскрывается в главе 37 второго тома, это синтаксис объектов Unicode в Python 2.X. Поскольку текстовые строки Unicode поддерживают все обычные строковые операции, разница для кода незначительна, пока текст находится в памяти; она становится наиболее важной при передаче текста в файлы и обратно и, как правило, только для разновидностей текста, отличающихся от ASCII, где в игру вступают кодировки.



В мире Python имеется также поддержка трансляции объектов в XML – текстового формата, применяемого в главе 37 второго тома (детали ищите в веб-сети). Другим частично связанным инструментом, который имеет дело с файлами форматированных данных, является стандартный библиотечный модуль `csv`. Он разбирает и создает данные CSV (comma-separated value – значения, разделенные запятыми) в файлах и строках. Он не отображается напрямую на объекты Python, но представляет собой еще один распространенный формат обмена данными:

```
>>> import csv  
>>> rdr = csv.reader(open('csvdata.txt'))  
>>> for row in rdr: print(row)  
...  
['a', 'bbb', 'cc', 'dddd']  
['11', '22', '33', '44']
```

Хранение упакованных двоичных данных: модуль `struct`

Прежде чем двигаться дальше, уместно сделать одно замечание относительно файлов: некоторые расширенные приложения также нуждаются в работе с упакованными двоичными данными, возможно созданными программой на языке С или сетевым подключением. В состав стандартной библиотеки Python входит инструмент, призванный помочь в этой области – модулю `struct` известно, как формировать и разбирать упакованные двоичные данные. В определенном смысле он является еще одним инструментом преобразования данных, который интерпретирует строки в файлах как двоичные данные.

Обзор модуля `struct` был приведен в главе 4, но давайте взглянем на него по-другому. Чтобы создать файл с упакованными двоичными данными, его необходимо открыть в режиме `'wb'` (двоичная запись) и передать `struct` строку формата, а также объекты Python. Используемая ниже строка формата обозначает пакет как 4-байтовое целое число, 4-символьную строку (которая должна быть строкой `bytes`, начиная с Python 3.2) и 2-байтовое целое число, все в форме с обратным порядком байтов (другие коды формирования поддерживают дополняющие байты, числа с плавающей точкой и т.д.):

```
>>> F = open('data.bin', 'wb')                      # Открытие двоичного выходного файла  
>>> import struct  
>>> data = struct.pack('>i4sh', 7, b'spam', 8)    # Создание упакованных  
                                                # двоичных данных  
>>> data  
b'\x00\x00\x00\x07spam\x00\x08'  
>>> F.write(data)                                    # Запись строки байтов  
>>> F.close()
```

Python создает строку двоичных данных `bytes`, которая записывается в файл обычным образом. Файл состоит в основном из непечатаемых символов, выводимых в виде шестнадцатеричных управляющих последовательностей, и является обычно-

венным двоичным файлом, как те, что встречались ранее. Чтобы разобрать значения в нормальные объекты Python, мы просто читаем строку из файла и распаковываем ее с применением той же самой строки формата. Python извлекает значения в стандартные объекты — целые числа и строку:

```
>>> F = open('data.bin', 'rb')
>>> data = F.read()                                # Получение упакованных двоичных данных
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> values = struct.unpack('>i4sh', data) # Преобразование в объекты Python
>>> values
(7, b'spam', 8)
```

Файлы двоичных данных — это развитые и отчасти низкоуровневые инструменты, которые мы не будем здесь раскрывать более подробно; дополнительную информацию можно получить в главе 37, где рассматривается модуль `struct`, в руководстве по библиотеке Python или интерактивно с помощью функции `help`, передав ей импортированный модуль `struct`. Также важно отметить, что вы можете использовать двоичные режимы '`wb`' и '`rb`' для обработки более простого двоичного файла, хранящего изображение или аудиозапись, как единого целого, не распаковывая его содержимое; в таких случаях ваш код может передавать его без разбора другим файлам либо инструментам.

Диспетчеры контекстов для файлов

Вас также заинтересует обсуждение поддержки диспетчеров контекстов для файлов в главе 34, которая появилась в версиях Python 3.0 и 2.6. Хотя такая поддержка больше относится к средствам обработки исключений, чем к самим файлам, она позволяет поместить код обработки файла внутрь уровня логики, который гарантирует, что файл будет закрыт (а его буферы вывода сброшены на диск в случае необходимости) автоматически при выходе вместо того, чтобы полагаться на автоматическое закрытие во время сборки мусора:

```
with open(r'C:\code\data.txt') as myfile:      # Детали приводятся в главе 34
    for line in myfile:
        ...обработка строк файла...
```

Оператор `try/finally`, также обсуждаемый в главе 34, способен предоставить аналогичную функциональность, но за счет дополнительного кода — точнее трех добавочных строк (тем не менее, мы часто можем избежать обоих вариантов и разрешить Python закрывать файлы автоматически):

```
myfile = open(r'C:\code\data.txt')
try:
    for line in myfile:
        ...обработка строк файла...
finally:
    myfile.close()
```

Схема диспетчера контекста `with` обеспечивает освобождение системных ресурсов во всех версиях Python и может оказаться более удобной для гарантирования сброса буферов выходных файлов; однако в отличие от более универсального оператора `try` она также ограничена объектами, которые поддерживают ее протокол. Но поскольку оба варианта требуют большего объема информации, чем получено к настоящему моменту, мы рассмотрим их позже в книге.

Другие файловые операции

Существуют дополнительные, более специализированные файловые методы, перечисленные в табл. 9.2, а также те, которые там не описаны. Например, как упоминалось ранее, метод `seek` переустанавливает текущую позицию в файле (следующая операция чтения или записи произойдет в новой позиции), `flush` принудительно записывает буферизированный вывод на диск, не закрывая файл (по умолчанию файлы всегда буферизированы), и т.д.

В руководстве по стандартной библиотеке Python и справочниках, описанных в предисловии, приводятся полные списки файловых методов; для быстрой справки вызовите `dir` или `help` интерактивно, передав открытый файловый объект (в Python 2.X, но не в Python 3.X, взамен можно передать имя `file`). Добавочные примеры обработки файлов предлагаются во врезке “Что потребует внимания: приемы просмотра файлов” в главе 13. В ней в общих чертах описаны часто употребляемые шаблоны кода для циклов просмотра файлов, содержащие операторы, которые пока еще не рассматривались, чтобы их можно было применить здесь.

Кроме того, имейте в виду, что хотя функция `open` и возвращаемые ею файловые объекты являются главным интерфейсом к внешним файлам в сценарии Python, в инструментальном наборе Python доступны дополнительные средства. Ниже перечислены некоторые из них.

Стандартные потоки данных

Предварительно открытые файловые объекты в модуле `sys`, такие как `sys.stdout` (см. раздел “Операции вывода” в главе 11).

Дескрипторные файлы в модуле os

Целочисленные файловые дескрипторы, которые поддерживают средства более низкого уровня, такие как блокировка файлов (см. режим `x` в функции `open` из Python 3.3 для эксклюзивного создания).

Сокеты, конвейеры и очереди FIFO

Похожие на файлы объекты, используемые для синхронизации процессов или взаимодействия через сети.

Файлы с доступом по ключу, поддерживаемые модулем shelf

Применяются для хранения неизмененных объектов Python с помощью модуля `pickle` напрямую по ключу (см. главу 28).

Потоки данных командной оболочки

Инструменты вроде `os.popen` и `subprocess.Popen`, которые поддерживают порождение команд оболочки и чтение/запись в их стандартные потоки данных (см. главы 13 и 21).

Область с открытым кодом предлагает еще больше инструментов, подобных файлам, в том числе поддержку взаимодействия с последовательными портами в расширении `PySerial` и интерактивных программ в системе `repect`. За дополнительными сведениями об инструментах, похожих на файлы, обращайтесь к книгам, ориентированным на приложения Python, или поищите информацию в веб-сети.



Примечание, касающееся нестыковки версий. В Python 2.X встроенное имя `open` по существу является синонимом для имени `file`, и формально открывать файлы можно путем вызова либо `open`, либо `file` (хотя `open` в целом предпочтительнее). В Python 3.X имя `file` больше не доступно из-за своей избыточности по отношению к `open`.

Пользователи Python 2.X могут также использовать имя `file` в качестве типа файлового объекта для настройки файлов с помощью объектно-ориентированного программирования (рассматриваемого позже в книге). В Python 3.X файлы радикально изменились. Классы, применяемые для реализации файловых объектов, находятся в стандартном библиотечном модуле `io`. Просмотрите документацию по модулю `io` или код классов, которые он делает доступными для настройки, и вызовите `type(F)` на открытом файле `F` для получения подсказок.

Обзор и сводка по основным типам

Теперь, когда вы видели все основные типы Python в действии, давайте завершим наше путешествие по объектам обзором свойств, которые они разделяют. В табл. 9.3 приведена классификация всех крупных типов, показанных до сих пор, согласно введенным ранее категориям типов. Далее описан ряд моментов, которые необходимо запомнить.

- Объекты разделяют операции в соответствии со своими категориями; например, объекты последовательностей – строки, списки и кортежи – разделяют операции над последовательностями, такие как конкатенация, определение длины и индексация.
- Только изменяемые объекты – списки, словари и множества – могут быть модифицированы на месте; изменять на месте числа, строки или кортежи не разрешено.
- Файлы экспортят только методы, поэтому изменяемость в действительности к ним неприменима – состояние файлов может меняться при их обработке, но это не совсем то же, что и ограничения изменяемости основных типов Python.
- “Числа” в табл. 9.3 включают все числовые типы: целое (и отдельное длинное целое в Python 2.X), с плавающей точкой, комплексное, десятичное и дробь.
- “Строки” в табл. 9.3 включают `str`, а также `bytes` в Python 3.X и `unicode` в Python 2.X; строковый тип `bytearray` в Python 3.X, 2.6 и 2.7 является изменяемым.
- Множества кое в чем похожи на ключи словаря без значений, но они не отображаются на значения и не упорядочены, поэтому не являются ни типом отображения, ни типом последовательности; `frozenset` – неизменяемый вариант `set`.
- В дополнение к операциям для категорий типов, начиная с версий Python 2.6 и 3.0, все типы в табл. 9.3 имеют вызываемые методы, которые обычно специфичны для своих типов.

Таблица 9.3. Классификация объектов

Тип объекта	Категория	Изменяемый?
Числа (все)	Числовые	Нет
Строки (все)	Последовательности	Нет
Списки	Последовательности	Да
Словари	Отображения	Да
Кортежи	Последовательности	Нет
Файлы	Расширения	–
Множества	Множества	Да
frozenset	Множества	Нет
bytearray	Последовательности	Да

Что потребует внимания: перегрузка операций

В части VI книги будет показано, что объекты, которые реализуются с помощью классов, можно произвольно выбирать из указанных в табл. 9.3 категорий. Скажем, если мы хотим предоставить новый вид специализированного объекта последовательности, согласованного со встроенными последовательностями, то можем написать код класса, который перегружает операции вроде индексации и конкатенации:

```
class MySequence:
    def __getitem__(self, index):
        # Вызывается для self[index]
    def __add__(self, other):
        # Вызывается для self + other
    def __iter__(self):
        # Предпочтителен в итерациях
```

и т.д. Мы также можем сделать новый объект изменяемым или нет, выборочно реализуя методы, которые вызываются для операций изменения на месте (например, `__setitem__` вызывается для присваиваний `self[index]=value`). Несмотря на то что данная тема выходит за рамки книги, новые объекты можно также реализовывать на внешнем языке, подобном C, как типы расширений C. Для этого мы заполняем места указателей на функции C, чтобы выбирать между наборами операций над числами, последовательностями и отображениями.

Гибкость объектов

В настоящей части книги было представлено несколько составных типов объектов – коллекций с компонентами. В общем случае:

- списки, словари и кортежи способны содержать в себе объекты любых видов;
- множества могут содержать неизменяемые объекты любых видов;
- списки, словари и кортежи допускают вложение на произвольную глубину;
- списки, словари и множества способны динамически увеличиваться и уменьшаться.

Поскольку составные типы объектов Python поддерживают произвольные структуры, они хорошо подходят для представления сложной информации в программах. Например, значения в словарях могут быть списками, содержащими кортежи, которые могут содержать словари, и т.д. Разрешено настолько глубокое вложение, насколько требуется для моделирования данных, подлежащих обработке.

Давайте рассмотрим пример вложения. В следующем взаимодействии определяется дерево вложенных составных объектов последовательностей, показанное на рис. 9.1. Для доступа к его компонентам можно включать любое необходимое количество операций индексации. Python оценивает индексы слева направо и на каждом шаге извлекает ссылку на более глубоко вложенный объект. Структура данных на рис. 9.1 может быть патологически сложной, но она иллюстрирует синтаксис, в общем случае используемый для доступа к вложенным объектам:

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```

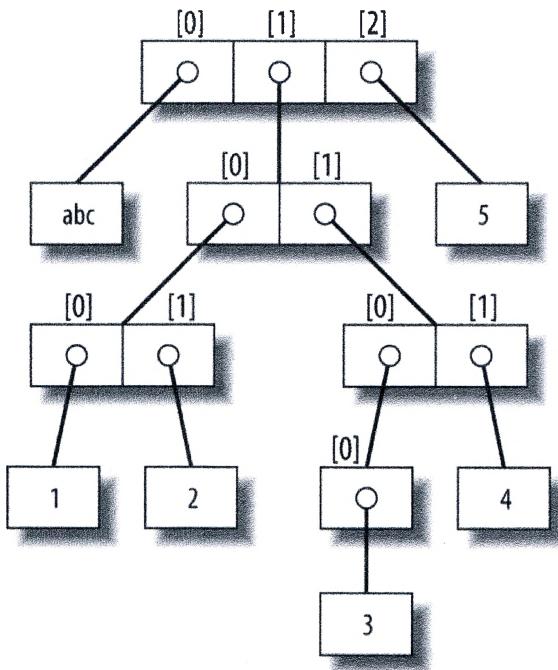


Рис. 9.1. Дерево вложенных объектов со смещениями его компонентов, созданное за счет выполнения лiteralного выражения ['abc', [(1, 2), ([3], 4)], 5]. Синтаксически вложенные объекты внутренне представляются как ссылки (т.е. указатели) на отдельные участки памяти

Ссылки или копии

В главе 6 упоминалось, что присваивания всегда сохраняют ссылки на объекты, а не их копии. На практике обычно так и требуется. Тем не менее, поскольку присваивания способны создавать множество ссылок на один и тот же объект, важно осознавать, что модификация изменяемого объекта на месте может затрагивать другие ссылки на тот же объект внутри программы. Если такое поведение нежелательно, тогда понадобится сообщить Python о необходимости явного копирования объекта.

Мы изучали это явление в главе 6, но оно может стать более тонким, когда в игру вступают крупные объекты вроде тех, что исследовались позже. Скажем, в следующем примере создается один список, который присваивается X, и другой список, присваиваемый L, который включает ссылку на список X. Вдобавок создается словарь D, содержащий еще одну ссылку на список X:

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']          # Встроенные ссылки на объект X
>>> D = {'x':X, 'y':2}
```

В данный момент есть три ссылки на созданный первый список: из имени X, изнутри списка, присвоенного L, и изнутри словаря, присвоенного D. Ситуация изображена на рис. 9.2.

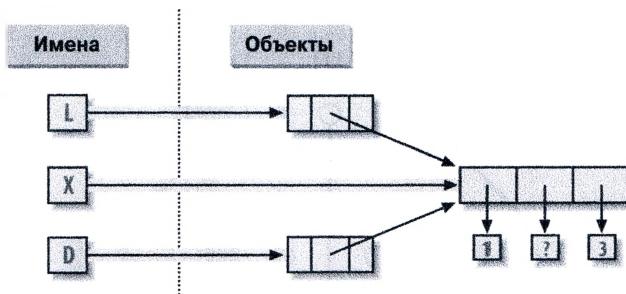


Рис. 9.2. Разделяемые ссылки на объекты: из-за того, что на список в X ссылаются также переменные L и D, изменение этого совместно используемого списка через переменную X делает его другим для L и D

Так как списки изменяемы, модификация разделяемого спискового объекта через любую из трех ссылок также отражается на остальных двух ссылках:

```
>>> X[1] = 'surprise'      # Изменяет все три ссылки!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y': 2}
```

Ссылки являются более высокоуровневым аналогом указателей в других языках, по которым всегда производится следование, когда они используются. Хотя вы не можете заполучить саму ссылку, одну и ту же ссылку допускается сохранять в нескольких местах (переменных, списках и т.д.). Это полезная характеристика — вы можете передавать крупный объект внутри программы, не создавая попутно его копии, что со-пряжено с затратами. Однако если вам действительно нужны копии, тогда вы можете запросить их:

- выражения срезов с пустыми границами (`L[:]`) копируют последовательности;
- метод `copy` словаря, множества и списка (`X.copy()`) копирует словарь, множество и список (метод `copy` доступен в списках, начиная с версии Python 3.3);
- некоторые встроенные функции, такие как `list` и `dict`, делают копии (`list(L)`, `dict(D)`, `set(S)`);
- стандартный библиотечный модуль `copy` при необходимости создает полные копии.

Например, пусть имеется список и словарь, и вы не хотите, чтобы их значения изменялись через другие переменные:

```
>>> L = [1, 2, 3]
>>> D = {'a':1, 'b':2}
```

Для этого просто присвойте другим переменным копии, а не ссылки на данные объекты:

```
>>> A = L[:] # Вместо A = L (или list(L))
>>> B = D.copy() # Вместо B = D (то же самое для множеств)
```

В таком случае изменения, сделанные через другие переменные, будут воздействовать на копии, а не на оригинал:

```
>>> A[1] = 'Ni'
>>> B['c'] = 'spam'
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Ni', 3], {'a': 1, 'c': 'spam', 'b': 2})
```

Что касается первоначального примера, то избежать побочных эффектов от ссылок можно путем указания среза исходного списка вместо его имени:

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b'] # Встраивание копий объекта X
>>> D = {'x':X[:], 'y':2}
```

В итоге картина на рис. 9.2 меняется – `L` и `D` теперь указывают на другие списки, отличающиеся от того, на который указывает `X`. Конечный результат заключается в том, что изменения, производимые через `X`, будут влиять только на `X`, но не на `L` и `D`; аналогично изменения в `L` или `D` не окажут воздействие на `X`.

Последнее замечание относительно копий: срезы с пустыми границами и словарный метод `copy` создают только копии *верхнего уровня*, т.е. не копируют вложенные структуры данных, если они присутствуют. Когда требуется полная независимая копия структуры данных с глубоким вложением (вроде различных структур записей из предшествующих глав), применяйте стандартный библиотечный модуль `copy`, представленный в главе 6:

```
import copy
X = copy.deepcopy(Y) # Полная копия объекта Y с произвольно глубоким вложением
```

Вызов `copy.deepcopy` рекурсивно обходит объекты с целью копирования всех их частей. Тем не менее, такая ситуация возникает гораздо реже – вот почему для использования данной схемы приходится писать больше кода. Как правило, ссылки будут тем, что необходимо; если это не так, то срезы и метод `copy` скопируют столько, сколько обычно требуется для работы.

Сравнения, равенство и истинность

Все объекты Python также реагируют на сравнения: проверки на предмет равенства, проверки относительных величин и т.п. Сравнения Python всегда инспектируют все части составных объектов, прежде чем может быть определен результат. На самом деле, когда присутствуют вложенные объекты, Python автоматически обходит структуры данных для применения сравнений слева направо и настолько глубоко, насколько нужно. Первое отличие, обнаруженнное по пути, и определит результат сравнения.

Такое действие иногда называют *рекурсивным* сравнением – одно и то же сравнение, запрошенное на объектах верхнего уровня, применяется ко всем вложенным объектам, ко всем их вложенным объектам и т.д., пока не будет найден результат. Позже в книге (в главе 19) вы увидите, как писать собственные рекурсивные функции, которые работают похожим образом на вложенных структурах. А пока подумайте о сравнении всех связанных страниц на двух веб-сайтах, если вас интересует модель таких структур и причина написания рекурсивных функций для их обработки.

В отношении основных типов рекурсия происходит автоматически. Например, сравнение списковых объектов автоматически сравнивает все их компоненты до тех пор, пока не будет найдено отличие или не будет достигнут конец:

```
>>> L1 = [1, ('a', 3)]      # Одинаковые значения, уникальные объекты
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2    # Эквивалентны? Тот же самый объект?
(True, False)
```

Здесь L1 и L2 присвоены списки, которые являются эквивалентными, но отдельными объектами. Как кратко объяснялось в главе 6, из-за природы ссылок Python существуют два способа проверки на равенство.

- Операция `==` проверяет эквивалентность значений. Python выполняет проверку эквивалентности, рекурсивно сравнивая все вложенные объекты.
- Операция `is` проверяет идентичность объектов. Python проверяет, являются ли два объекта на самом деле одним и тем же объектом (т.е. располагаются по тому же самому адресу в памяти).

В предыдущем примере L1 и L2 проходят проверку `==` (они имеют эквивалентные значения, потому что все их компоненты эквивалентны), но не проходят проверку `is` (они ссылаются на два разных объекта и, следовательно, на два разных участка памяти). Однако взгляните, что происходит для коротких строк:

```
>>> S1 = 'spam'
>>> S2 = 'spam'
>>> S1 == S2, S1 is S2
(True, True)
```

У нас снова должны быть два отдельных объекта, которые имеют одно и то же значение: проверка `==` должна дать `True`, а проверка `is` – `False`. Но поскольку Python внутреннее кеширует и повторно использует некоторые строки с целью оптимизации, в действительности в памяти присутствует только одна строка `'spam'`, разделяемая переменными S1 и S2, поэтому проверка идентичности `is` сообщит результат `True`. Чтобы получить нормальное поведение, нам понадобятся более длинные строки:

```
>>> S1 = 'a longer string'
>>> S2 = 'a longer string'
>>> S1 == S2, S1 is S2
(True, False)
```

Разумеется, по причине *неизменяемости* строк механизм кеширования никак не связан с вашим кодом – строки нельзя модифицировать на месте вне зависимости от того, сколько переменных на них ссылается. Если проверки идентичности выглядят сбивающими с толку, тогда загляните в главу 6, чтобы освежить в памяти концепции ссылок на объекты.

В качестве эмпирического правила запомните: операция `==` является тем, что вы будете использовать для почти всех проверок эквивалентности, тогда как операция `is` зарезервирована для очень специализированных ролей. Позже в книге будут приведены случаи применения обеих операций.

Сравнения относительных величин также работают рекурсивно в отношении вложенных структур данных:

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2    # Меньше, равно, больше: кортеж результатов
(False, False, True)
```

Здесь `L1` больше `L2`, потому что вложенное значение 3 больше, чем 2. К этому времени вы должны знать, что результатом последней строки на самом деле будет кортеж из трех объектов – результатов трех введенных выражений (пример кортежа без круглых скобок).

Говоря более конкретно, Python сравнивает типы следующим образом.

- Числа сравниваются по относительным величинам после преобразования в случае необходимости к общему наибольшему типу.
- Строки сравниваются лексикографически (по значениям кодовых точек из таблицы символов, возвращаемым функцией `ord`) символ за символом до достижения конца или первого несовпадения ("abc" < "ac").
- Списки и кортежи сравниваются путем сравнения каждого компонента слева направо и рекурсивно для вложенных структур до достижения конца или первого несовпадения (`[2] > [1, 2]`).
- Множества равны, если оба содержат те же самые элементы (формально, если каждое является подмножеством другого), а сравнения множеств по относительным величинам применяют проверки на подмножество и надмножество.
- Словарии считаются равными, если равны их отсортированные списки пар (ключ, значение). Сравнения по относительным величинам для словарей не поддерживаются в Python 3.X, но работают в Python 2.X, как если бы сравнивались отсортированные списки пар (ключ, значение).
- Сравнения величин разнородных типов (например, `1 < 'spam'`) вызывают ошибку в Python 3.X. Они разрешены в Python 2.X, но используют фиксированное и вместе с тем произвольное правило, основанное на строке имени типа. Опосредованно то же самое применимо к сортировке, которая внутренне использует сравнения: коллекции нечисловых разнородных типов не могут быть отсортированы в Python 3.X.

В общем случае сравнения структурированных объектов происходят так, как если бы мы записали объекты в виде литералов и сравнивали все их части по одной за раз слева направо. В последующих главах вы увидите другие типы объектов, которые способны изменять способ своего сравнения.

Сравнения и сортировка разнородных типов в Python 2.X и Python 3.X

Согласно последнему пункту списка из предыдущего раздела изменение в Python 3.X для сравнений нечисловых разнородных типов применяется к проверкам величин, а не равенства, но оно также опосредованно применяется к *сортировке*, которая внутренне выполняет проверки величин. В Python 2.X все это работает, хотя разнородные типы сравниваются путем произвольного упорядочения:

```
c:\code> c:\python27\python
>>> 11 == '11'          # Проверка равенства не преобразует нечисловые значения
False
>>> 11 >= '11'        # В Python 2.X сравниваются строки имен типов: int, str
False
>>> ['11', '22'].sort() # То же самое в отношении сортировки
>>> [11, '11'].sort()
```

Но в Python 3.X проверка величин разнородных типов запрещена за исключением числовых и вручную преобразованных типов:

```
c:\code> c:\python33\python
>>> 11 == '11'          # Python 3.X: проверка равенства работает,
                           # но проверка величин - нет
False
>>> 11 >= '11'
TypeError: unorderable types: int() > str()
Ошибка типа: неупорядочиваемые типы: int() > str()

>>> ['11', '22'].sort() # То же самое в отношении сортировки
>>> [11, '11'].sort()
TypeError: unorderable types: str() < int()
Ошибка типа: неупорядочиваемые типы: str() < int()

>>> 11 > 9.123 # Числа разнородных типов преобразуются в наибольший тип
True
>>> str(11) >= '11', 11 >= int('11') # Проблему решают ручные преобразования
(True, True)
```

Сравнения словарей в Python 2.X и Python 3.X

Второй с конца пункт списка из предыдущего раздела также заслуживает иллюстрации. В Python 2.X словари поддерживают сравнения величин, как если бы сравнивались отсортированные списки пар “ключ/значение”:

```
C:\code> c:\python27\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2      # Равенство словарей: Python 2.X + Python 3.X
False
>>> D1 < D2      # Сравнение величин для словарей: только Python 2.X
True
```

Тем не менее, как кратко упоминалось в главе 8, сравнения величины для словарей были изъяты из Python 3.X, потому что они влекут за собой слишком высокие накладные расходы, когда необходимо лишь выяснить равенство (при проверке равенства в Python 3.X используется оптимизированная схема, которая не сравнивает буквально отсортированные списки пар “ключ/значение”):

```
C:\code> c:\python33\python
>>> D1 = {'a':1, 'b':2}
```

```

>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
TypeError: unorderable types: dict() < dict()
Ошибка типа: неупорядочиваемые типы: dict() < dict()

```

Альтернативный подход в Python 3.X предусматривает либо написание циклов для сравнения значений по ключу, либо сравнение отсортированных списков пар “ключ/значение” вручную – словарного метода `items` и встроенной функции `sorted` будет достаточно:

```

>>> list(D1.items())
[('b', 2), ('a', 1)]
>>> sorted(D1.items())
[('a', 1), ('b', 2)]
>>>
>>> sorted(D1.items()) < sorted(D2.items()) # Проверка величин в Python 3.X
True
>>> sorted(D1.items()) > sorted(D2.items())
False

```

Подход связан с написанием большего объема кода, но на практике в большинстве программ, требующих такого поведения, будут разрабатываться более эффективные способы для сравнения данных в словарях, чем описанный выше обходной прием либо исходное поведение в Python 2.X.

Смысл понятий “истина” и “ложь” в Python

Обратите внимание, что результаты проверок, возвращаемые в последних двух примерах, представляют значения истины и лжи. Они выводятся как слова `True` и `False`, но теперь, когда мы основательно пользуемся логическими проверками вроде показанных, мне следует быть чуть более формальным в отношении фактического смысла этих имен.

Как и в большинстве языков программирования, в Python целое число 0 представляет ложь, а целое число 1 – истину. Однако в дополнение Python идентифицирует любую пустую структуру данных как ложь, а любую непустую – как истину. В более общем смысле понятия “истина” и “ложь” являются внутренними характеристиками *каждого* объекта в Python – любой объект будет либо истиной, либо ложью, как описано ниже:

- числа представляют собой ложь в случае нуля и истину в противном случае;
- остальные объекты представляют собой ложь, если они пусты, и истину в противном случае.

В табл. 9.4 приведены примеры значений истины и лжи объектов в Python.

Таблица 9.4. Примеры значений истинности объектов

Объект	Значение	Объект	Значение
"spam"	True	{}	False
""	False	1	True
[1, 2]	True	0.0	False
[]	False	None	False
{'a': 1}	True		

В качестве одного приложения, поскольку объекты сами представляют собой истину или ложь, обычное дело видеть, что программисты на Python пишут код проверок наподобие `if X:`, который при условии, что `X` – строка, делает такое же действие, как и код `if X != ''`. Другими словами, для выяснения, содержит ли объект что-нибудь, разрешено проверять его самого вместо того, чтобы сравнивать объект с пустым и потому представляющим ложь объектом того же типа (операторы `if` рассматриваются в следующей главе).

Объект `None`

В последней строке табл. 9.4 было показано, что в Python также предлагается специальный объект по имени `None`, который всегда считается ложью. Объект `None` кратко упоминался в главе 4; он представляет собой единственное значение специального типа данных в Python и обычно служит пустым заполнителем (во многом похоже на указатель `NULL` в С).

Например, вспомните, что присваивать по смещению в списках нельзя, если смещение не существует – список не увеличивается волшебным образом при попытке присваивания за его границами. Чтобы заранее выделить память под 100-элементный список и предоставить возможность работы с элементами по любому из 100 смещений, можно заполнить его объектами `None`:

```
>>> L = [None] * 100
>>>
>>> L
[None, None, None, None, None, None, None, ...]
```

Размер списка тем самым не ограничивается (он по-прежнему способен увеличиваться и уменьшаться), а просто предварительно устанавливается начальный размер, чтобы позволить выполнять будущие присваивания по индексам. Конечно, таким же способом список можно было бы инициализировать нулями, но рекомендуемая практика требует применения `None`, если тип содержимого списка варьируется или пока не известен.

Имейте в виду, что `None` не означает “неопределенный”. То есть `None` представляет собой что-то, а не ничего (несмотря на свое имя (`none` – ничто!)) – это реальный объект и реальный участок памяти, который создается и назначается встроенному имени самим Python. Позже в книге будут демонстрироваться другие использования специального объекта `None`; как вы узнаете в части IV, он также по умолчанию возвращается из функций, которые не завершают свою работу выполнением оператора `return` с результирующим значением.

Тип `bool`

Хотя мы исследуем тему истинности, также примите к сведению, что булевский тип `bool` в Python, введенный в главе 5, только дополняет понятия “истина” и “ложь”. Как было показано в главе 5, встроенные слова `True` и `False` являются просто настроенными версиями целых чисел `1` и `0` – как если бы данным двум словам заранее присваивались `1` и `0` повсюду в Python. Из-за особенностей реализации этого нового типа на самом деле он лишь слегка расширяет уже описанные понятия “истина” и “ложь” и направлен на то, чтобы сделать значения истинности более явными:

- при использовании в коде проверки истинности слова `True` и `False` эквиваленты `1` и `0`, но они проясняют намерение программиста;
- результаты булевых проверок, выполняемые интерактивно, выводятся как слова `True` и `False`, а не числа `1` и `0`, чтобы прояснить тип результата.

Вы не обязаны применять только булевский тип в логических операторах вроде `if`; все объекты по своему существу представляют собой истину или ложь, и если вы используете другие типы, то все булевские концепции, упомянутые в настоящей главе, по-прежнему работают, как здесь описано. Python также предлагает встроенную функцию `bool`, которая может применяться для проверки булевского значения объекта, когда это желательно сделать явным (например, является ли истинным, т.е. ненулевым или непустым):

```
>>> bool(1)
True
>>> bool('spam')
True
>>> bool({})
False
```

Тем не менее, на практике вы редко будете обращать внимание на булевский тип, производимый логическими проверками, поскольку булевские результаты автоматически используются операторами `if` и другими инструментами выбора. Мы продолжим исследовать булевские значения при изучении логических операторов в главе 12.

Иерархии типов Python

В качестве сводки и справки на рис. 9.3 приведена упрощенная схема всех встроенных типов Python и отношений между ними. Мы рассмотрели наиболее заметные из них; большинство других видов объектов на рис. 9.3 соответствует программным единицам (например, функции и модули) или открытым внутренним свойствам интерпретатора (скажем, стековые фреймы и скомпилированный код).

Здесь важнее всего отметить, что в системе Python *абсолютно все* является типом объекта и может обрабатываться вашими программами на Python. Например, вы можете передавать класс функции, присваивать его переменной, помещать его в список или словарь и т.д.

Объекты `type`

В действительности в Python даже сами типы представляют собой объекты: тип объекта является объектом типа `type`. Строго говоря, вызов встроенной функции `type(X)` возвращает объект типа для объекта X. Практическое применение этого заключается в том, что объекты типов могут использоваться для ручных сравнений типов в операторах `if` языка Python. Однако по причинам, которые приводились в главе 4, проверка типов вручную обычно не считается правильным действием в Python, т.к. она ограничивает гибкость кода.

Одно замечание по поводу имен типов: начиная с версии Python 2.2, каждый основной тип имеет встроенное имя, добавленное для поддержки настройки типов через объектно-ориентированное создание производных классов: `dict`, `list`, `str`, `tuple`, `int`, `float`, `complex`, `bytes`, `type`, `set` и т.д. В Python 3.X эти имена ссылаются на классы, а в Python 2.X, но не в Python 3.X, имя `file` также является именем типа и синонимом для функции `open`. Обращения к указанным именам на самом деле представляют собой вызовы конструкторов объектов, а не просто функций преобразования, хотя при элементарном применении вы можете трактовать их как простые функции.

Вдобавок стандартный библиотечный модуль `types` в Python 3.X предлагает дополнительные имена типов для типов, которые не доступны в виде встроенных (скажем, тип функции; в Python 2.X, но не в Python 3.X, модуль `types` также включает синонимы для имен встроенных типов), и делает возможными проверки типов с помощью функции `isinstance`.

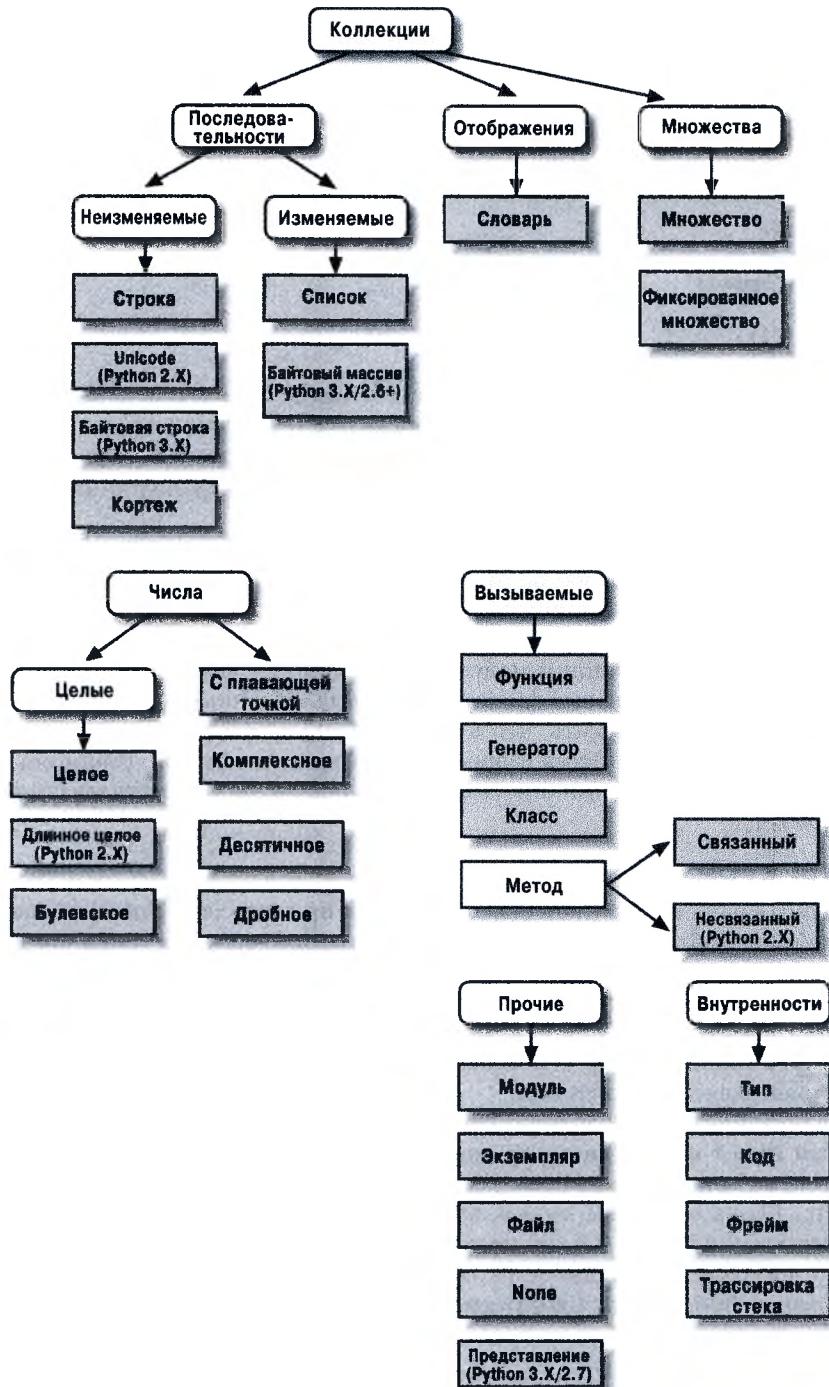


Рис. 9.3. Главные встроенные типы объектов Python, организованные по категориям. В Python абсолютно все является типом объекта, даже сам тип объекта! Некоторые типы расширений, такие как именованные кортежи, также могут относиться к этой диаграмме, но критерии для включения в набор основных типов не являются формальными!

Например, все показанные далее проверки типов дают `True`:

```
type([1]) == type([])      # Сравнение с типом другого списка
type([1]) == list          # Сравнение с именем спискового типа
isinstance([1], list)      # Проверка, список ли это или его настройка
import types                # Модуль types содержит имена для других типов
def f(): pass
type(f) == types.FunctionType
```

Поскольку теперь типы в Python можно создавать как производные классы, рекомендуется использовать методику с функцией `isinstance`. Создание производных классов для встроенных типов в Python 2.2 и последующих версиях более подробно обсуждается в главе 32 второго тома.



В главе 32 мы также исследуем, как `type(X)` и проверка типов в целом применяются к экземплярам *классов*, определяемых пользователем. Выражаясь кратко, в Python 3.X и для классов нового стиля в Python 2.X типом экземпляра класса является класс, из которого был создан экземпляр. В случае традиционных классов Python 2.X экземпляры всех классов взамен относятся к типу “экземпляр” и для содержательного сравнения типов мы должны сравнивать атрибуты `__class__` экземпляров. Так как вы пока еще не готовы заняться темой классов, мы отложим ее до главы 32.

Прочие типы в Python

Помимо основных объектов, обсуждаемых в этой части книги, и объектов программных единиц, таких как функции, модули и классы, которые мы встретим позже, обычная установка Python располагает десятками дополнительных типов объектов, доступных в форме связанных расширений С или классов Python – объекты регулярных выражений, файлы DBM, виджеты графических пользовательских интерфейсов, сетевые сокеты и т.д. В зависимости от того, у кого вы спрашиваете, рассмотренный ранее в главе *именованный кортеж* также может быть отнесен к данной категории (`Decimal` и `Fraction` из главы 5, как правило, более неоднозначны).

Главное отличие между такими дополнительными инструментами и встроенными типами, которые мы видели до сих пор, заключается в том, что для встроенных типов в языке предусмотрен специальный синтаксис создания их объектов (например, 4 для целого числа, [1, 2] для списка, функция `open` для файлов и `def` и `lambda` для функций). Прочие инструменты, как правило, доступны в стандартных библиотечных модулях, которые придется сначала импортировать для использования, и обычно не считаются основными типами. Скажем, чтобы создать объект регулярного выражения, понадобится импортировать модуль `re` и вызвать `re.compile()`. Исчерпывающее руководство по всем инструментам, доступным для программ Python, ищите в справочнике по библиотеке Python.

Затруднения, связанные со встроенными типами

Мы подошли к концу исследования основных типов данных. В завершение этой части книги мы обсудим распространенные проблемы, с которыми могут столкнуться новички (а временами даже эксперты), и предложим их решения. Кое-что в обсуждении является обзором тех идей, которые уже были раскрыты, но проблемы достаточно важны для того, чтобы напомнить о них еще раз.

Присваивание создает ссылки, а не копии

По причине крайней важности концепции я упомяну о ней снова: разделяемые ссылки на изменяемые объекты в программе могут иметь значение. Например, в следующем взаимодействии на списоковый объект, присвоенный имени L, производится ссылка из L и изнутри списка, присвоенного имени M. Изменение L на месте также изменяет то, на что ссылается M:

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']      # Встраивание ссылки на L
>>> M
['X', [1, 2, 3], 'Y']
>>> L[1] = 0                # Изменяет также и M
>>> M
['X', [1, 0, 3], 'Y']
```

Это следствие становится важным только в более крупных программах, к тому же разделяемые ссылки часто представляют собой именно то, что нужно. Если объекты изменяются нежелательным образом, тогда вы можете избежать применения разделяемых ссылок, явно копируя объекты. В случае списка всегда можно создать его копию верхнего уровня с использованием среза с пустыми границами, а также других методик, описанных ранее:

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']    # Встраивание копии L (или list(L), или L.copy())
>>> L[1] = 0                # Изменяет только L, но не M
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

Вспомните, что границы среза по умолчанию принимаются равными 0 и длине нарезаемой последовательности; если обе границы не указаны, то срез извлекает все элементы из последовательности, тем самым создавая копию верхнего уровня (новый, неразделяемый объект).

Повторение добавляет один уровень глубины

Повторение последовательности похоже на добавление ее к самой себе несколько раз. Тем не менее, в случае вложения изменяемых последовательностей результат может не всегда быть тем, который ожидался. Скажем, в следующем примере X присваивается список L, повторенный четыре раза, тогда как Y присваивается список, который содержит список L, повторенный четыре раза:

```
>>> L = [4, 5, 6]
>>> X = L * 4            # Подобно [4, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4          # [L] + [L] + ... = [L, L, ...]
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Поскольку во втором повторении L вкладывается, в Y попадают встроенные ссылки на исходный список, присвоенный L, и потому возникают побочные эффекты того же рода, что и описанные в предыдущем разделе:

```
>>> L[1] = 0          # Воздействует на Y, но не на X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

Сценарий может показаться искусственным или учебным, но до тех пор, пока неожиданно не случится в вашем коде! Здесь применимы те же самые решения проблемы, как и в предыдущем разделе, потому что фактически это лишь еще один способ создания ситуации с разделяемыми ссылками на изменяемые объекты – создавайте копии, когда разделяемые ссылки нежелательны:

```
>>> L = [4, 5, 6]
>>> Y = [list(L)] * 4      # Встраивание (разделяемой) копии L
>>> L[1] = 0
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Есть даже более тонкий момент: хотя имя `Y` больше не разделяет объект с `L`, оно по-прежнему встраивает четыре ссылки на одну и ту же копию объекта. Чтобы избежать и такого разделения, можно обеспечить уникальность каждой встроенной копии:

```
>>> Y[0][1] = 99      # Все четыре копии по-прежнему одинаковы
>>> Y
[[4, 99, 6], [4, 99, 6], [4, 99, 6], [4, 99, 6]]
>>> L = [4, 5, 6]
>>> Y = [list(L) for i in range(4)]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
>>> Y[0][1] = 99
>>> Y
[[4, 99, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Если вы вспомните, что повторение, конкатенация и нарезание копируют только верхний уровень объектов своих операндов, тогда ситуации подобного рода обретут гораздо больший смысл.

Остерегайтесь циклических структур данных

Фактически мы столкнулись с этой концепцией в предыдущем примере: если объект коллекции содержит ссылку на самого себя, то он называется *циклическим объектом*. Всякий раз, когда Python обнаруживает цикл в объекте, он выводит [...] вместо того, чтобы застрять в бесконечном цикле (как когда-то давно):

```
>>> L = ['grail']        # Добавление ссылки на тот же самый объект
>>> L.append(L)         # Создает цикл в объекте: [...]
>>> L
['grail', [...]]
```

Кроме понимания, что три точки в квадратных скобках представляют цикл в объекте, о таком случае надлежит знать, поскольку он способен приводить к затруднениям – циклические структуры могут стать причиной того, что ваш код попадет в неожиданные циклы, если вы не предупредите их.

Например, определенные программы, проходящие через структурированные данные, должны хранить список, словарь или множество уже посещенных элементов и проверять его, когда собираются войти в цикл, который мог бы вызвать нежелатель-

ную петлю. Дополнительную информацию о проблеме можно почерпнуть из решения упражнений части I в приложении. Также дождитесь общего обсуждения рекурсии в главе 19, программы `reloadall.py` в главе 25 и класса `ListTree` в главе 31 второго тома, где будут приведены конкретные примеры программ, в которых обнаружение циклов имеет значение.

Решение в знании: не используйте циклические ссылки, если только действитель но не нуждаешься в них, и удостоверьтесь в том, что предупреждаете их в программах, где они могут возникать. Существуют веские причины создавать циклы, но в отсутствие кода, которому известно, как их обрабатывать, ссылающиеся сами на себя объекты могут стать больше неожиданностью, нежели чем-то полезным.

Неизменяемые типы нельзя модифицировать на месте

И еще раз ради завершенности: нельзя модифицировать на месте неизменяемый объект. Взамен понадобится создать новый объект с помощью нарезания, конкатенации и т.п. и при необходимости присваивать его исходной ссылке:

```
T = (1, 2, 3)
T[2] = 4           # Ошибка!
T = T[:2] + (4,)  # Нормально: (1, 2, 4)
```

Это выглядит как дополнительная работа по написанию кода, но положительный момент в том, что описанные выше затруднения не могут возникнуть, когда применяются неизменяемые объекты, подобные кортежам и строкам. Поскольку неизменяемые объекты нельзя модифицировать на месте, с ними не связаны побочные эффекты, которым подвержены списки.

Резюме

В главе рассматривались последние два крупных типа основных объектов — кортежи и файлы. Вы узнали, что кортежи поддерживают все обычные операции над последовательностями, имеют несколько методов, не разрешают модификацию на месте по причине неизменяемости и расширяются типом именованных кортежей. Также вы узнали о том, что файловые объекты возвращаются встроенной функцией `open` и предлагают методы для чтения и записи данных.

Попутно мы выяснили, как транслировать объекты Python в и из строк для сохранения в файлах, и взглянули на модули `pickle`, `json` и `struct`, предназначенные для решения более сложных задач (серIALIZАЦИЯ объектов и двоичные данные). В заключение был приведен обзор ряда характеристик, общих для объектов всех типов (например, разделяемые ссылки), и перечень распространенных затруднений в области, связанной с типами объектов.

С следующей части книги мы перейдем к теме *синтаксиса операторов* — способа кодирования логики обработки в сценариях. Кроме того, будут исследованы все базовые процедурные операторы. В следующей главе эта тема открывается введением в общую модель синтаксиса Python, которая применима ко всем видам операторов. Однако прежде чем двигаться дальше, необходимо ответить на контрольные вопросы главы и затем проработать упражнения, предлагаемые в конце части, для закрепления концепций типов данных. Операторы по большому счету всего лишь создают и обрабатывают объекты, так что перед тем, как переходить к следующей части, проверьте, хорошо ли вы усвоили типы данных, выполнив упражнения.

Проверьте свои знания: контрольные вопросы

1. Как определить размер кортежа? Почему этот инструмент находится там, где он есть?
2. Напишите выражение, которое изменяет первый элемент в кортеже. В процессе кортеж `(4, 5, 6)` должен стать `(1, 5, 6)`.
3. Что принимается по умолчанию для аргумента режима обработки в вызове `open`?
4. Какой модуль вы бы использовали для сохранения объектов Python в файле без предварительного преобразования их в строки?
5. Как бы вы копировали сразу все части вложенной структуры?
6. Когда Python считает объект истинным?
7. В чем заключается ваша цель?

Проверьте свои знания: ответы

1. Встроенная функция `len` возвращает длину (количество содержащихся элементов) для любого контейнерного объекта в Python, включая кортежи. `len` является встроенной функцией, а не методом типа, потому что она применяется к множеству разных типов объектов. В общем случае встроенные функции и выражения могут охватывать много типов объектов; методы специфичны для одного типа, хотя некоторые могут быть доступны для нескольких типов (например, `index` работает на списках и кортежах).
2. Поскольку кортежи неизменяемы, их нельзя модифицировать на месте, но можно создать новый кортеж с желаемым значением. При наличии `T = (4, 5, 6)` изменить первый элемент можно за счет создания нового кортежа из частей имеющегося кортежа с помощью нарезания и конкатенации: `T = (1,) + T[1:]`. (Вспомните, что одноэлементные кортежи требуют хвостовой запятой.) Можно было бы также преобразовать кортеж в список, модифицировать его на месте и преобразовать обратно в кортеж, но такой прием сопряжен с более высокими затратами и редко требуется на практике – если известно, что объект потребуется изменять на месте, тогда просто необходимо использовать список.
3. По умолчанию для аргумента режима обработки в вызове `open` принимается `'r'`, т.е. чтение текстового ввода. Для входных текстовых файлов нужно просто передать имя внешнего файла.
4. Для сохранения объектов Python в файле без их явного преобразования в строки можно применять модуль `pickle`. Модуль `struct` является родственным, но предполагает, что данные должны быть представлены внутри файла в упакованном двоичном формате; модуль `json` аналогичным образом преобразует ограниченный набор объектов Python в из строки согласно формату JSON.
5. Когда необходимо скопировать все части вложенной структуры `X`, следует импортировать модуль `copy` и вызвать `copy.deepcopy(X)`. На практике это случается редко; обычно желаемым поведением являются ссылки, и в большинстве случаев, как правило, достаточно поверхностных копий (например, `aList[:]`, `aDict.copy()`, `set(aSet)`).

6. Объект считается истинным, если он представляет собой либо ненулевое число, либо непустую коллекцию. Встроенные слова True и False по существу предопределены, чтобы означать то же, что и целые числа 1 и 0.
7. Подходящими ответами могут быть “Изучить Python”, “Перейти к чтению следующей части книги”, “Найти Святой Грааль”.

Проверьте свои знания: упражнения для части II

Здесь вам предлагается пройтись по основам встроенных объектов. Как и ранее, попутно могут возникать новые идеи, поэтому по завершении проработки упражнений обязательно просмотрите их решения в приложении. В случае нехватки времени я рекомендую начать с упражнений 10 и 11 (самых практических из всех) и затем заняться остальными, если время позволяет. Ввиду фундаментальности материала постарайтесь выполнить столько упражнений, сколько сумеете; программирование – практическая деятельность, и ничто не заменит опробование прочитанного на практике, чтобы закрепить понимание идей.

1. *Основы.* Поэкспериментируйте интерактивно с распространенными операциями, приведенными в различных таблицах в этой части книги. Для начала запустите интерактивный интерпретатор Python, наберите каждое из приведенных ниже выражений и попробуйте объяснить, что происходит в каждом случае. Обратите внимание, что точка с запятой используется в качестве разделителя операторов для размещения нескольких операторов в одной строке: скажем, `X=1; X` присваивает и затем выводит переменную (синтаксис операторов будет рассматриваться в следующей части книги). Также помните, что запятая между выражениями обычно создает кортеж даже в отсутствие окружающих круглых скобок: `X, Y, Z` – это трехэлементный кортеж, который Python выводит в круглых скобках.

```
2 ** 16
2 / 5, 2 / 5.0
"spam" + "eggs"
S = "ham"
"eggs " + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)
'green {0} and {1}'.format('eggs', S)
('x',)[0]
('x', 'y')[1]
L = [1,2,3] + [4,5,6]
L, L[::-1], L[:-2], L[-2:]
([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
L.reverse(); L
L.sort(); L
L.index(4)
{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
D['w'] = 0
D['x'] + D['w']
```

```
D[(1,2,3)] = 4
list(D.keys()), list(D.values()), (1,2,3) in D
[], ["", (), {}], None
```

2. *Индексация и нарезание.* В интерактивной подсказке определите список по имени L, который содержит четыре строки или числа (например, L=[0,1,2,3]). Затем поэкспериментируйте со следующими граничными случаями. Вы можете никогда не встретить их в реальных программах (особенно в таком причудливом виде, как приведено здесь). Тем не менее, они заставят вас задуматься о лежащей в основе модели, а некоторые окажутся полезными в менее искусственных формах – скажем, нарезание за границами может помочь, если последовательность настолько длинная, насколько ожидалось.

- a) Что происходит, когда вы пытаетесь индексировать за границами (например, L[4])?
 - б) Как насчет нарезания за границами (например, L[-1000:100])?
 - в) Наконец, каким образом Python поступит, если вы попытаетесь извлечь последовательность в обратном направлении, когда нижняя граница больше верхней границы (например, L[3:1])? Подсказка: попробуйте выполнить присваивание этому срезу (L[3:1] = ['?']) и посмотрите, куда было помещено значение. Как вы думаете, это может быть тем же самым явлением, которое вы наблюдали при нарезании за границами?
3. *Индексация, нарезание и оператор del.* Определите другой список L с четырьмя элементами и присвойте по одному из его смещений пустой список (например, L[2]=[]). Что произошло? Затем присвойте срезу пустой список (L[2:3]=[]). Что теперь произошло? Вспомните, что присваивание срезу удаляет его и вставляет новое значение туда, где раньше был срез.

Оператор del удаляет по смещениям, ключам, атрибутам и именам. Примените его к списку, чтобы удалить элемент (например, del L[0]). Что произойдет, если вы удалите целый срез (del L[1:])? Что произойдет, когда вы присвоите срезу объект, не являющийся последовательностью (L[1:2]=1)?

4. *Присваивание кортежам.* Наберите следующие строки:

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
```

Как вы думаете, что происходит с X и Y, когда вы набираете эти строки?

5. *Ключи словарей.* Рассмотрим такой фрагмент:

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
```

Вы узнали, что словари не поддерживают доступ по смещению, так что же здесь происходит? Проливает ли свет на происходящее следующий фрагмент? (Подсказка: к какой категории типов относятся строки, целые числа и кортежи?)

```
>>> D[(1, 2, 3)] = 'c'
>>> D
{(1, 2, 3): 'c'}
```

6. *Индексация словарей.* Создайте словарь по имени D с тремя элементами для ключей 'a', 'b' и 'c'. Что происходит, когда вы пытаетесь произвести индексацию для несуществующего ключа (D['d'])? Что делает Python, если вы пробуете выполнить присваивание по несуществующему ключу 'd' (например, D['d']='spam')? Как это соотносится с присваиваниями и ссылками за границами для списков? Выглядит ли это похожим на правило для имен переменных?
7. *Универсальные операции.* Запустите интерактивные проверки, чтобы ответить на перечисленные далее вопросы.
- Что случается, когда вы пытаетесь использовать операцию + на отличающихся/разнородных типах (например, строка + список, список + кортеж)?
 - Работает ли операция +, когда один из операндов является словарем?
 - Работает ли метод append для списков и строк? Как насчет использования метода keys на списках? (Подсказка: что именно метод append допускает об объекте, к которому применяется?)
 - Какой тип объекта вы получаете обратно при выполнении нарезания или конкатенации двух списков или двух строк?
8. *Индексация строк.* Определите строку S из четырех символов: S = "spam". Затем наберите выражение S[0][0][0][0]. Можете объяснить, что происходит на этот раз? (Подсказка: вспомните, что строка является коллекцией символов, но символы Python представляют собой односимвольные строки.) Будет ли такая индексация работать в случае применения к списку, подобному ['s', 'p', 'a', 'm']? Почему?
9. *Неизменяемые типы.* Снова определите строку S из четырех символов: S = "spam". Напишите выражение присваивания, которое изменяет строку S на "slam", используя только нарезание и конкатенацию. Можно ли было бы выполнить ту же самую операцию с применением только индексации и конкатенации? Как насчет присваивания по индексу?
10. *Вложение.* Напишите структуру данных, которая представляет вашу персональную информацию: имя (имя, отчество, фамилия), возраст, место работы, адрес, адрес электронной почты и номер телефона. Можете создавать структуры данных с помощью любой желаемой комбинации встроенных типов объектов (списки, кортежи, словари, строки, числа). Затем получите доступ к индивидуальным компонентам созданных структур данных. Являются ли некоторые структуры для этого объекта более осмысленными, чем другие?
11. *Файлы.* Напишите сценарий, который создает новый выходной файл по имени myfile.txt и записывает в него строку "Hello file world!". Затем напишите еще один сценарий, который открывает myfile.txt, читает и выводит его содержимое. Запустите два сценария в командной строке системы. Появились ли новый файл в каталоге, где вы запускали свои сценарии? Что будет, если вы включите в имя файла, передаваемое функции open, другой путь к каталогу? Примечание: файловые методы write не добавляют в ваши строки символы новой строки; если вы хотите закончить строку в файле, тогда добавьте в ее конец явный символ \n.

ЧАСТЬ III

Операторы и синтаксис

Введение в операторы Python

Теперь, когда вы знакомы с основными встроенными типами объектов Python, мы приступаем к исследованию главных форм операторов Python. Как и в предыдущей части, мы начнем с обобщенного введения в синтаксис операторов, после чего в последующих нескольких главах более подробно рассмотрим специфические операторы.

Попросту говоря, *операторы* – это то, что вы пишете для сообщения Python о том, что должны делать ваши программы. Если согласно главе 4 программы “делают дела с помощью оснащения”, тогда операторы являются способом указания вида *дел*, которые делает программа. Более формально Python представляет собой процедурный, основанный на операторах язык; за счет комбинирования операторов вы задаете *процедуру*, которую Python выполняет для достижения целей программы.

Еще раз о концептуальной иерархии Python

Другой способ постижения роли операторов предусматривает мысленный возврат к концептуальной иерархии, введенной в главе 4, где речь шла о встроенных объектах и выражениях, используемых для манипулирования ими. В настоящей главе иерархия поднимается на следующий уровень структуры программ Python.

1. Программы состоят из модулей.
2. Модули содержат операторы.
3. Операторы содержат выражения.
4. Выражения создают и обрабатывают объекты.

В своей основе программы, написанные на языке Python, состоят из операторов и выражений. Выражения обрабатывают объекты и встраиваются в операторы. Операторы кодируют более крупную логику работы программы – они применяют и направляют выражения для обработки объектов, которые вы изучали в предшествующих главах. Более того, операторы представляют собой место, где появляются объекты (например, в выражениях внутри операторов присваивания), а некоторые операторы создают совершенно новые виды объектов (функции, классы и т.д.). На верхнем уровне операторы всегда существуют в модулях, которые сами управляются с помощью операторов.

Операторы Python

В табл. 10.1 приведена сводка по набору операторов Python. Каждый оператор Python имеет собственное специфическое назначение и собственный специфический *синтаксис* – правила, определяющие его структуру, – хотя, как мы увидим, многие разделяют общие синтаксические шаблоны, а роли ряда операторов перекрываются. В табл. 10.1 также даны примеры каждого оператора, закодированные в соответствии с его синтаксическими правилами. В ваших программах такие единицы кода могут выполнять действия, повторять задачи, делать выбор, создавать более крупные программные структуры и т.д.

В настоящей части книги обсуждаются операторы с первой строки таблицы вплоть до `break` и `continue`. Вам уже неформально было представлено несколько операторов из табл. 10.1; в этой части книги мы восполним опущенные ранее детали, ознакомим с остатком набора процедурных операторов Python и раскроем полную синтаксическую модель. Операторы, расположенные ниже в табл. 10.1, которые касаются более крупных программных единиц – функций, классов, модулей и исключений – подводят к более крупным понятиям программирования, так что каждый из них будет рассматриваться в отдельном разделе. Более специализированные операторы (вроде `del`, который удаляет разнообразные компоненты) раскрываются в других местах книги, а также описаны в стандартных руководствах по Python.

Таблица 10.1. Операторы Python

Оператор	Роль	Пример
Присваивания	Создание ссылок	<code>a, b = 'good', 'bad'</code>
Вызовы и другие выражения	Выполнение функций	<code>log.write("spam, ham")</code>
Вызовы <code>print</code>	Вывод объектов	<code>print('The Killer', joke)</code>
<code>if/elif/else</code>	Выбор действий	<code>if "python" in text: print(text)</code>
<code>for/else</code>	Итерация	<code>for x in mylist: print(x)</code>
<code>while/else</code>	Универсальные циклы	<code>while X > Y: print('hello')</code>
<code>pass</code>	Пустой заполнитель	<code>while True: pass</code>
<code>break</code>	Выход из цикла	<code>while True: if exittest(): break</code>
<code>continue</code>	Продолжение цикла	<code>while True: if skiptest(): continue</code>
<code>def</code>	Функции и методы	<code>def f(a, b, c=1, *d): print(a+b+c+d[0])</code>
<code>return</code>	Результаты функций	<code>def f(a, b, c=1, *d): return a+b+c+d[0]</code>
<code>yield</code>	Генераторные функции	<code>def gen(n): for i in n: yield i*2</code>

Оператор	Роль	Пример
global	Пространства имен	x = 'old' def function(): global x, y; x = 'new'
nonlocal	Пространства имен (Python 3.X)	def outer(): x = 'old' def function(): nonlocal x; x = 'new'
import	Доступ к модулям	import sys
from	Доступ к атрибутам	from sys import stdin
class	Построение объектов	class Subclass(Superclass): staticData = [] def method(self): pass
try/except/ finally	Перехват исключений	try: action() except: print('action error')
raise	Генерация исключений	raise EndSearch(location)
assert	Отладочные проверки	assert X > Y, 'X too small'
with/as	Диспетчеры контекста (Python 3.X, 2.6+)	with open('data') as myfile: process(myfile)
del	Удаление ссылок	del data[k] del data[i:j] del obj.attr del variable

Формально в табл. 10.1 воспроизведены операторы Python 3.X. Хотя этого перечня операторов достаточно для обзора и справочных целей, в том виде, как есть, он не совсем полон. Ниже описано несколько тонкостей относительно его содержимого.

- Операторы присваивания имеют различные синтаксические формы, описанные в главе 11: базовая, последовательности, дополненная и др.
- Формально `print` в Python 3.X не является ни зарезервированным словом, ни оператором, а вызовом встроенной функции; но поскольку `print` почти всегда будет выполняться в виде оператора с выражением (и часто занимать отдельную строку), такой вызов в общем случае трактуется как разновидность оператора. Операции вывода рассматриваются в главе 11.
- Начиная с Python 2.5, `yield` – также выражение, а не оператор; подобно `print` оно обычно используется как оператор с выражением и потому включено в табл. 10.1, но в главе 20 мы увидим, что сценарии иногда присваивают или по-другому применяют его результат. Будучи выражением, `yield` также является ключевым словом в отличие от `print`.

Большая часть табл. 10.1 применима и к Python 2.X за исключением случаев, когда это не так – если вы используете Python 2.X, то ниже приведено несколько замечаний, которые его касаются.

- В Python 2.X оператор `nonlocal` недоступен; как будет показано в главе 17, существуют альтернативные способы достижения того же эффекта сохранения перезаписываемого состояния.
- В Python 2.X `print` представляет собой оператор, а не вызов встроенной функции, со специфическим синтаксисом, раскрываемым в главе 11.
- В Python 2.X встроенная функция выполнения кода `exec` из Python 3.X является оператором со специфическим синтаксисом; однако поскольку он поддерживает окружающие круглые скобки, вы можете применять в коде Python 2.X форму вызова `exec`, принятую в Python 3.X.
- В Python 2.5 операторы `try/except` и `try/finally` были объединены: раньше они представляли собой два отдельных оператора, но теперь `except` и `finally` можно записывать в том же самом операторе `try`.
- В Python 2.5 оператор `with/as` является необязательным расширением, которое не будет доступным до тех пор, пока вы явно не включите его, выполнив оператор `from __future__ import with_statement` (см. главу 34).

История о двух `if`

Прежде чем мы погрузимся в детали конкретных операторов из табл. 10.1, я хочу начать обзор синтаксиса операторов Python, показав вам то, что вы *не* собираетесь вводить в коде Python. Это даст возможность сравнить синтаксис операторов Python с другими синтаксическими модулями, которые вы могли видеть в прошлом.

Взгляните на следующий оператор `if`, запрограммированный на С-подобном языке:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

Оператор может относиться к C, C++, Java, JavaScript или похожему языку. А вот эквивалентный оператор в языке Python:

```
if x > y:
    x = 1
    y = 2
```

Первое, что может броситься в глаза – эквивалентный оператор Python не настолько загроможден, т.е. в нем меньше синтаксических компонентов. Так было задумано; одна из целей Python как языка написания сценариев заключается в том, чтобы облегчить жизнь программистов, требуя меньшего объема набора.

В частности, сравнив две синтаксические модели, вы заметите, что Python добавляет один элемент к смеси, а три элемента, присутствующие в С-подобном языке, в коде Python отсутствуют.

Что Python добавляет

Новым компонентом синтаксиса в Python является символ двоеточия (`:`). Все *составные операторы* Python, т.е. операторы с вложенными другими операторами, собирают общий шаблон. Шаблон состоит из строки заголовка, завершающейся двоеточием, и вложенного блока кода, обычно указываемого с отступом под строкой заголовка:

Строка заголовка:

Вложенный блок операторов

Двоеточие обязательно, а его отсутствие является, пожалуй, наиболее распространенной ошибкой, допускаемой начинающими программистами на Python — безусловно, мне тысячи раз приходилось быть свидетелем такой ошибки в группах студентов, которых я обучал. На самом деле, если вы новичок в Python, вы почти наверняка вскоре забудете о символе двоеточия. В таком случае вы получите сообщение об ошибке, а большинство дружественных к Python редакторов позволят легко выявить такое недоразумение. Со временем включение двоеточия становится неосознанной привычкой (настолько сильной, что вы можете начать вводить двоеточия также и в коде на С-подобном языке, приводя к выдаче забавных сообщений об ошибках компилятором этого языка!).

Что Python устраниет

Несмотря на то что Python требует добавочного символа двоеточия, программистам на С-подобных языках приходится включать три элемента, которые обычно в Python не нужны.

Круглые скобки необязательны

Первый элемент — набор круглых скобок вокруг проверок в верхней части оператора:

```
if (x < y)
```

Круглые скобки здесь требуются синтаксисом многих С-подобных языков. Тем не менее, в Python они необязательны — мы просто опускаем круглые скобки, и оператор работает аналогичным образом:

```
if x < y
```

Говоря формально, из-за того, что любое выражение может быть помещено в круглые скобки, их наличие не повредит в таком коде Python, и они не трактуются как ошибка, когда присутствуют.

Но не поступайте так: вы будете без нужды изнашивать свою клавиатуру и заявлять всему миру о том, что являетесь программистом на С-подобном языке, все еще изучающим Python (я знаю, потому как сам был таким). “Образ действий Python” заключается в том, чтобы просто опускать круглые скобки в операторах такого вида.

Конец строки является концом оператора

Второй и более важный элемент синтаксиса, который вы не обнаружите в коде Python — точка с запятой. В Python вы не обязаны завершать операторы точками с запятой, как принято в С-подобных языках:

```
x = 1;
```

Общее правило в Python состоит в том, что конец строки автоматически завершает оператор, находящийся в этой строке. Другими словами, вы можете избавиться от точки с запятой, и оператор будет работать прежним образом:

```
x = 1
```

Как вы вскоре увидите, есть несколько способов обойти указанное правило (скажем, помещение кода внутрь структуры в квадратных скобках позволяет разнести ее на множество строк). Но в подавляющем большинстве кода вы будете записывать по одному оператору в строке, и никакие точки с запятой не потребуются.

Если вы тоскуете по тем дням, когда программировали на С (при условии, что такое вообще возможно), то можете продолжать использовать точки с запятой в конце каждого оператора — язык в состоянии смириться с их присутствием, поскольку точка с запятой служит также разделителем при комбинировании операторов.

Но не поступайте так тоже (серьезно!). Тем самым вы опять сообщаете миру о том, что являетесь программистом на С-подобном языке, который еще не полностью переключился на стиль написания кода Python. Стиль Python предусматривает отказ от точек с запятой. Судя по студентам в группах, некоторым программистам со стажем, похоже, нелегко отказаться от такой привычки. Но вы добьетесь своего; точки с запятой в роли завершения операторов — бесполезные помехи в Python.

Конец отступа является концом блока

Третий и последний элемент синтаксиса, устранивший в Python, отсутствие которого может выглядеть самым необычным для тех, кто недавно прекратил программировать на С-подобных языках, связан с тем, что вы не набираете в коде что-нибудь явное для синтаксической пометки начала и конца вложенного блока кода. Вам не нужно помещать вложенный блок внутрь `begin/end`, `then/endif` или фигурных скобок, как вы поступали бы в С-подобных языках:

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

В Python взамен мы согласованно смещаем все операторы в заданном одиночном вложенном блоке на одно и то же расстояние вправо, и для определения начала и конца блока Python применяет физические отступы операторов:

```
if x > y:  
    x = 1  
    y = 2
```

Под *отступом* здесь подразумевается пустое пространство слева от двух вложенных операторов. Python вовсе не заботит то, как сделан отступ (можно использовать либо пробелы, либо табуляции), или то, насколько отступ *большой* (допускается применять любое количество пробелов или табуляций). На самом деле отступ одного вложенного блока может совершенно отличаться от отступа другого. Синтаксическое правило лишь гласит о том, что все операторы заданного одиночного вложенного блока должны быть смещены на то же самое расстояние вправо. В противном случае возникает синтаксическая ошибка, а код не запустится до тех пор, пока вы не приведете отступы в согласованное состояние.

Для чего используется синтаксис с отступами?

Правило отступа может показаться на первый взгляд необычным программистам, привыкшим к С-подобным языкам, но это преднамеренная особенность Python и один из главных способов, которыми Python вынуждает программистов производить единообразный, систематический и читабельный код. Правило отступа по существу означает, что вы обязаны выравнивать свой код вертикально по столбцам в соответствии с его логической структурой. Совокупный эффект заключается в том, что код становится более согласованным и читабельным (в отличие от большинства кода на С-подобных языках).

Выражаясь более строго, выравнивание кода в соответствии с его логической структурой является значительной работой по содействию его читабельности и тем самым многократному использованию и удобству сопровождения, как вами, так и другими. В действительности, даже если вы не будете применять Python после прочтения книги, то все равно должны выработать привычку выравнивать свой код в целях читабельности в любом блочно-структурированном языке. Python акцентирует внимание на данной задаче, сделав выравнивание частью синтаксиса, но поступать так важно в любом языке программирования, что оказывает огромное влияние на полезность результирующего кода.

Ваш опыт может отличаться, но когда я все еще занимался разработкой на постоянной основе, мне главным образом платили за работу над крупными старыми программами C++, которые создавались многими программистами в течение нескольких лет. Почти у каждого программиста был свой стиль отступов в коде. Например, меня часто просили изменять цикл `while` в коде C++, который начинался примерно так:

```
while (x > 0) {
```

До того, как мы углубимся в отступы, следует отметить, что есть три или четыре способа, которыми программисты размещают фигурные скобки в C-подобном языке, и в организациях часто ведутся горячие споры и составляются руководства по стандартам, регламентирующие допустимые варианты (что выглядит более чем просто отклонением от задач, решаемых с помощью программирования). Как бы то ни было, вот вам сценарий, который я часто встречал в коде C++. Первый человек, работающий над кодом, использовал для тела цикла отступ в четыре пробела:

```
while (x > 0) {
    ----;
    ----;
```

Со временем этот человек перешел на руководящую должность, а его заменил другой человек, которому нравилось делать еще большие отступы вправо:

```
while (x > 0) {
    -----
    -----
    -----
    -----;
```

Позже любитель широких отступов воспользовался возможностью и сменил работу (положив конец царствованию кодового террора в своем лице...), а его место занял человек, который предпочитал делать поменьше отступов:

```
while (x > 0) {
    -----
    -----
    -----
    -----
    -----
    -----;
    -----;
}
```

И так далее. В конце концов, блок завершается закрывающей фигурной скобкой (}), которая, конечно же, делает его “блочно-структурированным кодом” (здесь присутствует изрядная доля сарказма). Нет: в любом блочно-структурированном языке, Python или еще каком, если вложенные блоки не имеют согласованных отступов, тогда их становится очень трудно истолковывать, изменять или повторно использовать, пото-

му что код больше визуально не отражает свой логический смысл. Читабельность имеет значение, и отступы являются важной составной частью читабельности.

Ниже приведен пример, на котором вы могли обжечься в прошлом, если много программировали на каком-то С-подобном языке. Взгляните на следующий оператор в C:

```
if (x)
    if (y)
        оператор1;
else
    оператор2;
```

К какому оператору `if` здесь относится `else`? Как ни удивительно, но в C часть `else` относится к вложенному оператору `if (if (y))`, несмотря на то, что визуально выглядит связанный с внешним оператором `if (x)`. Это классическая ловушка в языке C; она может привести к тому, что читатель совершенно неправильно интерпретирует код и некорректно изменяет его способами, которые иногда обнаруживаются лишь после того, как марсоход врезался в огромную скалу!

В Python подобное произойти не может — из-за того, что отступы существенны, код работает именно так, как выглядит. Вот эквивалентный оператор Python:

```
if x:
    if y:
        оператор1
else:
    оператор2
```

В приведенном примере часть `else` логически связана с оператором `if`, с которым выровнена по вертикали (внешний `if x`). В известной степени Python является языком WYSIWYG (what you see is what you get — что видишь, то и получаешь), поскольку внешний вид кода определяет путь его выполнения независимо от того, кто его писал.

Если сказанного по-прежнему не хватает, чтобы должностным образом оценить преимущества синтаксиса Python, то приведу еще один эпизод. В начале своей карьеры я работал в успешной компании, которая занималась разработкой системного программного обеспечения на языке C, где согласованные отступы не требовались. Даже в таких условиях, когда в конце дня мы сохраняли свой код в системе управления версиями исходного кода, в компании автоматически запускался сценарий, который анализировал отступы, применяемые в коде. Если сценарий замечал, что мы использовали отступы в коде несогласованно, то на следующее утро мы получали по электронной почте соответствующее сообщение — равно как и наши менеджеры!

Дело в том, что даже когда язык этого не требует, хорошим программистам известно, что согласованное применение отступов оказывает сильное влияние на читабельность и качество кода. Тот факт, что отступы в Python продвинуты до уровня синтаксиса, рассматривается большинством как особенность языка.

Также имейте в виду, что почти каждый дружественный к программистам текстовый редактор обладает встроенной поддержкой синтаксической модели Python. Скажем, в графическом пользовательском интерфейсе IDLE к строкам кода автоматически добавляется отступ при наборе вложенного блока; нажатие клавиши забоя возвращает обратно на один уровень отступов, и то, насколько далеко вправо IDLE смещает операторы во вложенном блоке, можно настраивать. Универсального стандарта не существует: общепринятыми являются четыре пробела или одна табуляция, но обычно вы сами решаете, каким образом и насколько отступать (если только не работаете в компании, где отступы регламентируются политикой и внутренними стан-

дартгами). Делайте больший отступ вправо для следующего вложенного блока и меньший отступ для закрытия предыдущего блока.

В качестве эмпирического правила запомните, что вероятно не стоит смешивать табуляции и пробелы в том же самом блоке в Python, если только не делать это согласованно; в отдельно взятом блоке используйте табуляции или пробелы, но не то и другое (на самом деле, как будет показано в главе 12, теперь Python 3.X сообщает об ошибке при несогласованном применении табуляций и пробелов). Более того, смешивать табуляции и пробелы в отступах, видимо, не следует в *любом* структурированном языке — такой код может вызвать крупные проблемы с читабельностью, если текстовый редактор у следующего программиста настроен на отображение табуляций не так, как у вас. С-подобные языки зачастую разрешают программистам обходить данное правило, но программисты не должны поступать так: результатом может оказаться изрядно запутанная смесь.

Независимо от языка, на котором пишется код, вы должны придерживаться согласованных отступов для обеспечения читабельности. Фактически, если вас не приучили делать это раньше, то ваши учителя оказали вам медвежью услугу. Большинство программистов (особенно те, кому приходится читать код, написанный другими) считает ценным качеством то, что Python продвигает правило, касающееся отступов, до уровня синтаксиса. Кроме того, инструментам, которые должны выводить код Python, не составляет труда генерировать табуляции вместо фигурных скобок. В целом, если вы делаете то, что в любом случае должны были бы делать в С-подобном языке, но избавляйтесь от фигурных скобок, тогда ваш код будет удовлетворять правилам синтаксиса Python.

Несколько специальных случаев

Как упоминалось ранее, в синтаксической модели Python:

- конец строки завершает оператор в этой строке (безо всяких точек с запятой);
- вложенные операторы объединяются в блок и ассоциируются согласно их физическим отступам (без фигурных скобок).

Указанные правила охватывают почти весь код Python, который вам доведется писать или встречать в реальности. Однако Python также предлагает ряд специализированных правил, которые делают возможной настройку как операторов, так и вложенных блоков операторов. Они не обязательны и должны использоваться умеренно, но программисты находят их полезными на практике.

Специальные правила для операторов

Хотя операторы обычно располагают по одному в строке, в Python разрешено помещать несколько операторов в одну строку, разделяя их точками с запятой:

```
a = 1; b = 2; print(a + b)      # Три оператора в одной строке
```

Это единственное место в Python, где требуются точки с запятой: в качестве *разделителей между операторами*. Тем не менее, такое применение допускается, только если объединяемые подобным образом операторы сами не являются составными. Другими словами, вы можете выстраивать в цепочку лишь простые операторы вроде присваиваний, вызовов print и обращений к функциям. Составные операторы, такие как проверки if и циклы while, по-прежнему должны находиться в собственных строках (иначе вы могли бы втиснуть всю программу в одну строку, что вряд ли добавило бы вам популярности среди коллег по работе!).

Другое специальное правило для операторов по существу является противоположностью: вы можете разнести одиночный оператор на *множество строк*. Для этого понадобится лишь поместить часть оператора внутрь пары скобок — круглых (()), квадратных ([]), или фигурных ({}). Любой код, заключенные в такие скобки, может занимать несколько строк: оператор не закончится до тех пор, пока Python не достигнет закрывающей скобки из пары. Например, вот как записать списковый литерал в нескольких строках:

```
mylist = [1111,  
          2222,  
          3333]
```

Поскольку код заключен в пару квадратных скобок, Python просто переходит на следующую строку до тех пор, пока не встретит закрывающую квадратную скобку. Фигурные скобки, окружающие словари (а также литералы множеств и включения словарей и множеств в Python 3.X/2.7), позволяют им распространяться на несколько строк, а круглые скобки делают это для кортежей, вызовов функций и выражений. Отступы в строках продолжения роли не играют, хотя здравый смысл подсказывает, что строки должны каким-то образом выравниваться ради читабельности.

Круглые скобки представляют собой универсальное средство — из-за того, что в них можно помещать любое выражение, вставка открывающей круглой скобки дает возможность продолжить оператор в следующей строке:

```
X = (A + B +  
      C + D)
```

Кстати, такая методика работает также и с составными операторами. Везде, где нужно записать крупное выражение, просто поместите его в круглые скобки, чтобы перенести на следующую строку:

```
if (A == 1 and  
    B == 2 and  
    C == 3):  
    print('spam' * 3)
```

Более старое правило позволяет продолжать оператор в следующей строке, если предыдущая строка заканчивается на обратную косую черту:

```
X = A + B + \  
    C + D # Подверженная ошибкам более старая альтернатива
```

Однако такая альтернативная методика вышла из употребления и в наши дни не одобряется, потому что замечать и сохранять обратные косые черты нелегко. Она также довольно хрупкая и подвержена ошибкам. Дело в том, что после обратной косой черты не должно быть пробелов, а случайный пропуск обратной косой черты может приводить к неожиданным эффектам, если следующая строка будет ошибочно воспринята как новый оператор. (В показанном выше примере C + D — сам по себе допустимый оператор, если он не имеет отступа.) Такое правило также является еще одним отголоском языка C, где оно обычно используется в макросах #define; если вы находитесь в мире Python, то и поступайте, как принято в Python, а не в C.

Специальные правила для блоков

Как упоминалось ранее, операторы во вложенном блоке кода обычно ассоциируются по их отступам на одно и то же расстояние вправо. В качестве одного специального случая здесь тело составного оператора может взамен находиться в той же самой строке, что и строка заголовка оператора Python, после двоеточия:

```
if x > y: print(x)
```

В результате у нас появляется возможность записывать однострочные операторы `if`, однострочные циклы `while` и `for` и т.д. Тем не менее, это будет работать, только если тело составного оператора само не содержит каких-либо составных операторов. То есть после двоеточия разрешено указывать лишь простые операторы – присваивания, вызовы `print`, обращения к функциям и т.п. Более крупные операторы по-прежнему должны находиться в собственных строках. Дополнительные части составных операторов (такие как часть `else` оператора `if`, который мы рассмотрим в следующем разделе) также обязаны располагаться в отдельных строках. Тела составных операторов могут состоять из множества простых операторов, разделенных точками с запятой, но обычно такой подход не одобряется.

В общем, хотя временами это необязательно, если вы будете размещать все операторы в отдельных строках и всегда делать отступы для вложенных блоков, то ваш код станет легче читать и изменять в будущем. Более того, некоторые инструменты для профилирования и покрытия кода могут быть не в состоянии проводить различия между множеством операторов, втиснутых в одну строку, или заголовком и телом однострочного составного оператора. Практически всегда в ваших интересах поддерживать в Python простоту. Вы можете применять специальные правила для написания кода Python, который трудно читать, но это требует немало работы, и у вас наверняка найдется, куда лучше потратить свое время.

Однако чтобы увидеть в действии простое и распространенное исключение из указанных правил (использование однострочного оператора `if` для выхода из цикла с помощью `break`), а также представить дополнительный синтаксис Python, в следующем разделе будет написан реальный код.

Короткий пример: интерактивные циклы

Мы увидим все описанные правила синтаксиса в действии, когда начнем тур по специфическим составным операторам Python в последующих нескольких главах, но повсюду в языке Python они работают одинаково. Давайте пока рассмотрим короткий, но реалистичный пример, который продемонстрирует практический способ совместного применения синтаксиса операторов и вложения операторов, а также попутно ознакомит с несколькими операторами.

Простой интерактивный пример

Предположим, что вас попросили написать на Python программу, которая взаимодействует с пользователем в окне консоли. Возможно, необходимо принимать ввод для отправки в базу данных или читать числа с целью использования в вычислении. Независимо от предназначения, вам придется написать код цикла, который читает одну или большее количество строк, набираемых пользователем на клавиатуре, и выводит для каждой результат. Другими словами, вы должны написать программу с классическим циклом чтение/оценка/вывод.

Типичный шаблонный код для такого цикла взаимодействия может выглядеть в Python следующим образом:

```
while True:  
    reply = input('Enter text:')  
    if reply == 'stop': break  
    print(reply.upper())
```

В коде применяется несколько новых идей и ряд тех, что вы уже видели ранее.

- В коде задействован цикл `while` языка Python – наиболее универсальный оператор цикла. Позже мы детально исследуем оператор `while`, а пока достаточно знать, что он состоит из слова `while`, за которым следует выражение, дающее истинный или ложный результат, а за ним вложенный блок кода, повторяющийся до тех пор, пока выражение остается истинным (слово `True` здесь считается всегда истинным).
- Встроенная функция `input`, которую мы встречали ранее в книге, используется для универсального консольного ввода – она выводит необязательный аргумент в качестве приглашения и возвращает то, что пользователь набрал, в виде строки. Согласно приведенной далее врезке “На заметку!” в Python 2.X взамен применяется `raw_input`.
- В коде также присутствует односрочный оператор `if`, который использует специальное правило для вложенных блоков: тело оператора `if` находится в строке заголовка после двоеточия вместо того, чтобы располагаться с отступом в строке ниже. В любом случае оператор будет работать одинаково, но так мы экономим одну строку.
- Наконец, оператор `break` языка Python применяется для немедленного выхода из цикла – происходит просто переход из цикла и программа продолжает выполнение сразу после оператора цикла. Без такого оператора выхода цикл `while` выполнялся бы бесконечно, потому что его проверка всегда истинна.

В сущности, приведенная комбинация операторов означает “читать строки, введенные пользователем, и выводить их в верхнем регистре до тех пор, пока пользователь не введет слово `stop`”. Существуют и другие способы написания такого цикла, но использованная здесь форма очень часто встречается в коде Python.

Обратите внимание, что все три строки, вложенные под строку заголовка `while`, смещены на одно и то же расстояние вправо – поскольку операторы подобным образом выровнены вертикально по столбцам, они являются блоком кода, который ассоциирован с проверкой `while` и повторяется. Для завершения блока с телом цикла будет достаточно либо конца файла исходного кода, либо оператора с меньшим отступом.

Когда код запускается, интерактивно или как файл сценария, вот какое взаимодействие мы получаем (весь код примера находится в файле `interact.py` из пакета примеров для этой книги):

```
Enter text:spam
SPAM
Enter text:42
42
Enter text:stop
```



Примечание, касающееся нестыковки версий. В примере написан код для Python 3.X. Если вы работаете в Python 2.X, то код работает так же, но во всех примерах текущей главы вместо `input` придется применять `raw_input`, и можно опустить внешние круглые скобки в операторах `print` (хотя никакого вреда они не наносят). В действительности, если вы исследуете файл `interact.py` из пакета примеров, то увидите, что в нем это делается автоматически – для поддержки совместимости с Python 2.X имя `input` переустанавливается, если старшим номером версии функционирующего Python является 2 (вызов `input` приводит к выполнению `raw_input`):

```
import sys
if sys.version[0] == '2': input = raw_input # Совместимость
# с Python 2.X
```

В Python 3.X имя `raw_input` было изменено на `input`, а `print` представляет собой встроенную функцию, а не оператор (более подробно `print` обсуждается в следующей главе). В Python 2.X также имеется оператор `input`, но он пытается выполнить входную строку, как если бы она было кодом Python, что вероятно не будет работать в данном контексте; в Python 3.X того же эффекта можно достичь посредством `eval(input())`.

Выполнение математических действий над пользовательским вводом

Наш сценарий работает, но теперь предположим, что вместо преобразования текстовой строки в верхний регистр мы хотим выполнить какие-то математические действия над числовым вводом — скажем, возвести его в квадрат, возможно в ошибочной попытке программы ввода возраста подразнить своих пользователей. Для получения желаемого результата мы могли бы использовать код вроде показанного ниже:

```
>>> reply = '20'
>>> reply ** 2
...текст сообщения об ошибке не показан...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Ошибка типа: неподдерживаемые типы операндов для ** или pow(): str и int
```

Тем не менее, прием в нашем сценарии работать не будет, потому что (как обсуждалось в предыдущей части книги) Python не преобразует типы в выражениях, если только все они не являются числовыми, а ввод от пользователя всегда возвращается сценарию в виде *строки*. Мы не сможем возвести строку цифр в степень до тех пор, пока вручную не преобразуем ее в целое число:

```
>>> int(reply) ** 2
400
```

Вооружившись этой информацией, мы можем переписать наш цикл для выполнения необходимого математического действия. Поместите следующий код в файл для его тестирования:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(int(reply) ** 2)
print('Bye')
```

Как и ранее, в сценарии применяется односрочный оператор `if` для выхода из цикла при вводе `stop`, но также осуществляется преобразование ввода для выполнения требуемого математического действия. В конце еще добавляется прощальное сообщение. Поскольку оператор `print` в последней строке не имеет такого же отступа, как у вложенного блока кода, он не считается частью тела цикла и будет выполнен только раз после выхода из цикла:

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```



Замечание по использованию. С этого момента я буду предполагать, что код хранится в файле сценария и запускается из него через командную строку, пункт меню IDLE или любым другим способом запуска файлов, которые обсуждались в главе 3. В пакете примеров файл называется `interact.py`. Однако если вы вводите этот код интерактивно, тогда удостоверьтесь в том, что включили пустую строку (т.е. два раза нажали клавишу `<Enter>`) перед финальным оператором `print`, чтобы завершить цикл. Это также подразумевает невозможность вырезания и вставки кода в интерактивную подсказку: добавочная пустая строка требуется в интерактивном режиме, но не в файле сценария. Хотя в наборе последнего оператора `print` в интерактивном режиме мало смысла — вы будете вводить его после взаимодействия с циклом!

Обработка ошибок путем проверки ввода

Пока все хорошо, но обратите внимание, что происходит в случае ввода недопустимой строки:

```
Enter text:xxx
...текст сообщения об ошибке не показан...
ValueError: invalid literal for int() with base 10: 'xxx'
Ошибка значения: недопустимый литерал для int() с основанием 10: xxx
```

Столкнувшись с ошибкой, встроенная функция `int` генерирует исключение. Если нужно, чтобы сценарий был надежным, тогда можно заранее проверить содержимое строки с помощью метода `isdigit` строкового объекта:

```
>>> S = '123'
>>> T = 'xxx'
>>> S.isdigit(), T.isdigit()
(True, False)
```

Это также дает повод к дальнейшему вложению операторов в примере. В приведенной далее обновленной версии нашего интерактивного сценария с применением полнофункционального оператора `if` исключение при ошибках обходится:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')
```

Оператор `if` подробно рассматривается в главе 12, но сейчас достаточно знать, что он представляет собой довольно легковесный инструмент для кодирования логики в сценариях. В своей полной форме оператор состоит из слова `if`, за которым следует проверка и связанный блок кода, одна или более необязательных проверок `elif` (“`else if`”) и блоков кода, а также необязательная часть `else` с ассоциированным блоком кода в качестве принимаемого по умолчанию. Python выполняет блок кода, связанный с первой проверкой, которая дает истину, работая сверху вниз, или часть `else`, если все проверки оказались ложными.

Части `if`, `elif` и `else` в предыдущем примере относятся к одному и тому же оператору, потому что все они выровнены по вертикали (т.е. используют одинако-

вый уровень отступа). Оператор `if` простирается от слова `if` до начала оператора `print` в последней строке сценария. В свою очередь весь блок `if` является частью цикла `while`, поскольку он располагается с отступом под строкой заголовка цикла. Подобного рода вложение операторов станет выглядеть естественным, как только вы освоитесь с ним.

После запуска нового сценария его код перехватывает ошибки до того, как они произойдут, и выводит сообщение, прежде чем продолжить работу (которое вероятно имеет смысл улучшить в более позднем выпуске), но ввод `stop` по-прежнему приводит к завершению, а допустимые числа возводятся в квадрат:

```
Enter text:5
25
Enter text:xyz
Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop
Bye
```

Обработка ошибок с помощью оператора `try`

Предыдущее решение работает, но позже в книге вы увидите, что самый универсальный способ обработки ошибок в Python предусматривает перехват и полное восстановление после них с применением оператора `try` языка Python. Мы будем исследовать этот оператор в части VII книги, а сейчас отметим, что использование `try` здесь может в итоге дать код, который многие сочтут более простым, чем предыдущая версия:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print('Bad!' * 8)
    else:
        print(num ** 2)
    print('Bye')
```

Новая версия кода работает в точности как предыдущая, но явную проверку на предмет ошибки мы заменили кодом, который предполагает успешное выполнение преобразования, и поместили его внутрь обработчика исключений для случаев, когда это не так. Другими словами, вместо обнаружения ошибки мы просто реагируем, когда она возникает.

Оператор `try` является еще одним составным оператором и следует такому же шаблону, как `if` и `while`. Он состоит из слова `try`, за которым находится основной блок кода (действие, подлежащее выполнению), затем часть `except` с кодом обработчика исключений и часть `else`, выполняемая в случае, если никакие исключения в части `try` не генерировались. Python сначала выполняет часть `try`, после чего либо часть `except` (если исключение произошло), либо часть `else` (если исключений не было).

С точки зрения вложения операторов, поскольку слова `try`, `except` и `else` имеют отступы на том же самом уровне, они считаются частью одного оператора `try`.



Здесь на месте `float` также работал бы вызов функции `eval` языка Python, который мы применяли в главах 5 и 9 для преобразования данных в строках и файлах. Тогда появилась бы возможность вводить произвольные выражения (`2 ** 100` было бы допустимым, хотя и необычным вводом, особенно с учетом того, что программа обрабатывает возраст!). Это мощная концепция, которая подвержена тем же самым проблемам с безопасностью, о которых шла речь в предшествующих главах. Если вы не можете доверять источнику строки с кодом, тогда используйте более ограничивающие инструменты преобразования наподобие `int` и `float`.

Функция `exec` языка Python, применяемая в главе 3 для выполнения кода из файла, похожа на `eval` (но предполагает, что строка является оператором, а не выражением, и не возвращает результат), а вызов `compile` предварительно компилирует часто используемые строки кода в объекты байт-кода ради достижения высокой скорости. Дополнительные сведения можно получить, запустив `help` для любого из указанных средств; как упоминалось ранее, `exec` представляет собой оператор в Python 2.X, но функцию в Python 3.X, поэтому в Python 2.X ищите ее описание в руководстве. Мы также будем применять `exec` в главе 25 для импортирования модулей, используя строку с именем (пример более динамических ролей данного инструмента).

Вложение кода на три уровня в глубину

Давайте рассмотрим последнее изменение кода. При необходимости вложение можно продолжить; скажем, мы могли бы расширить приведенный ранее сценарий, обрабатывающий только целые числа, для перехода к одной из набора альтернатив на основе относительной величины допустимого ввода:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        num = int(reply)
        if num < 20:
            print('low')
        else:
            print(num ** 2)
print('Bye')
```

В последней версии сценария добавлен оператор `if`, вложенный в часть `else` другого оператора `if`, который в свою очередь вложен в цикл `while`. Когда код является условным или повторяется как здесь, мы просто смещаем его дальше вправо. Совокупный эффект похож на предшествующие версии, но теперь для чисел, меньших 20, будет выводиться слово `low`:

```
Enter text:19
low
Enter text:20
400
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

Обратите внимание, что часть `else` здесь ассоциирована с `try`, а не с `if`. Как вы увидите, в Python часть `else` может появляться в операторах `if`, но также может присутствовать в операторах `try` и циклах – ее отступ сообщает, к какому оператору она относится. В данном случае оператор `try` начинается со слова `try` и охватывает код, расположенный с отступом под словом `else`, потому что `else` имеет такой же отступ, как у `try`. Оператор `if` в этом коде односторонний и заканчивается после `break`.

Поддержка чисел с плавающей точкой

Позже в книге мы еще вернемся к оператору `try`. Пока достаточно знать, что поскольку оператор `try` можно применять для перехвата любой ошибки, он сокращает объем кода проверки на предмет ошибок, который приходится писать, и представляет собой очень универсальный подход к работе с необычными случаями. Скажем, при наличии уверенности в том, что `print` не потерпит неудачу, тогда пример мог бы стать еще короче:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(int(reply) ** 2)
    except:
        print('Bad!' * 8)
    print('Bye')
```

И если мы хотим поддерживать ввод чисел с плавающей точкой вместо только целых, например, то использовать `try` было бы гораздо легче, чем вручную проверять на предмет ошибок – мы могли бы просто вызвать функцию `float` и перехватить ее исключения:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(float(reply) ** 2)
    except:
        print('Bad!' * 8)
    print('Bye')
```

На сегодняшний день функции вроде `isfloat` для строк не предусмотрено, поэтому такой подход на основе исключений избавляет нас от необходимости анализировать весь возможный синтаксис чисел с плавающей точкой посредством явной проверки на предмет ошибок. Когда код написан таким способом, мы можем вводить более разнообразные числа, но выявление ошибок и выход по-прежнему работают, как и ранее:

```
Enter text:50
2500.0
Enter text:40.5
1640.25
Enter text:1.23E-100
1.5129e-200
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

Резюме

На этом краткий обзор синтаксиса операторов Python завершен. В главе были представлены общие правила для написания операторов и блоков кода. Вы узнали, что в Python мы обычно записываем по одному оператору в строке и смещаем все операторы во вложенном блоке на одно и то же расстояние вправо (отступ – часть синтаксиса Python). Кроме того, мы также взглянули на ряд исключений из этих правил, в том числе строки продолжения и односторонние проверки и циклы. Наконец, мы воплотили все рассмотренные идеи на практике в сценарии с интерактивным циклом, где продемонстрировали несколько операторов и показали синтаксис операторов в действии.

В следующей главе мы начнем более тщательно исследовать каждый базовый процедурный оператор Python. Как вы увидите, все операторы соблюдают те же самые общие правила, которые были описаны в настоящей главе.

Проверьте свои знания: контрольные вопросы

1. Какие три элемента синтаксиса обязательны в С-подобном языке, но опущены в Python?
2. Как обычно завершается оператор в Python?
3. Как операторы во вложенном блоке обычно ассоциируются в Python?
4. Как можно было бы разместить одиночный оператор в нескольких строках?
5. Как можно было бы записать составной оператор в одной строке?
6. Есть ли веские причины набирать точку с запятой в конце оператора в Python?
7. Для чего предназначен оператор `try`?
8. Какую ошибку чаще всего допускают новички в Python?

Проверьте свои знания: ответы

1. С-подобные языки требуют наличия круглых скобок вокруг выражений проверки в некоторых операторах, точки с запятой в конце каждого оператора и фигурных скобок вокруг вложенного блока кода.
2. Конец строки завершает оператор, находящийся в этой строке. Если в одной строке присутствует несколько операторов, то их можно завершать с помощью точек с запятой; подобным же образом, если оператор охватывает множество строк, тогда его придется завершать путем помещения внутрь пары квадратных скобок.
3. Все операторы во вложенном блоке имеют отступы с одинаковым количеством табуляций или пробелов.
4. Разместить одиночный оператор в нескольких строках можно, поместив его часть в круглые, квадратные или фигурные скобки; оператор заканчивается, когда Python встречает строку, в которой содержится закрывающая скобка из пары.
5. Тело составного оператора может быть перемещено в строку заголовка после двоеточия, но только если тело состоит из несоставных операторов.
6. Лишь тогда, когда необходимо втиснуть несколько операторов в одну строку кода. И то это работает, только если операторы являются несоставными, к тому же не одобряется, поскольку может давать в итоге код, который трудно читать.
7. Оператор `try` применяется для перехвата и восстановления после исключений (ошибок) в сценарии Python. Он обычно выступает в качестве альтернативы ручной проверки на предмет ошибок в коде.
8. Самая распространенная ошибка новичков связана с тем, что они забывают набрать двоеточие в конце строки заголовка составного оператора. Если вы начали изучать Python и пока ее не допустили, то вероятно это скоро произойдет!

Операторы присваивания, выражений и вывода

После краткого введения в синтаксис операторов Python в этой главе начинаются углубленные исследования индивидуальных операторов Python. Первыми мы рассмотрим основы: операторы присваивания, операторы выражений и операторы вывода. Вы уже видели всех их в действии, но здесь мы восполним важные детали, опущенные до сих пор. Хотя они относительно просты, как вы заметите, для каждого вида операторов существуют необязательные вариации, которые пригодятся, когда вы приступите к написанию реальных программ на Python.

Операторы присваивания

Мы какое-то время использовали оператор присваивания Python для присваивания объектов именам. В его базовой форме слева от знака равенства записывается *цель присваивания*, а справа — присваиваемый *объект*. Целью слева может быть имя или компонент объекта, а объектом справа — произвольное выражение, результатом вычисления которого оказывается объект. Большой частью присваивания прямолинейны, но они обладают рядом характеристик, которые необходимо запомнить.

- **Присваивания создают ссылки на объекты.** Как обсуждалось в главе 6, присваивания в Python сохраняют в именах или компонентах структур данных ссылки на объекты. Они всегда создают ссылки на объекты, а не копируют объекты. По этой причине переменные Python больше похожи на указатели, чем на области хранения данных.
- **Имена создаются при первом присваивании.** Python создает имя переменной, когда ему в первый раз присваивается значение (т.е. ссылка на объект), так что нет нужды объявлять имена заранее. Во время присваивания также создаются некоторые (но не все) области структуры данных (например, элементы словарей, определенные атрибуты объектов и т.п.). После присваивания всякий раз, когда имя появляется в выражении, оно заменяется значением, на которое ссылается.

- Перед ссылкой именам должно быть выполнено присваивание. Ошибочно применять имя, которому пока еще не присваивалось значение. На попытку поступить так Python отреагирует генерацией исключения, а не возвращением какого-нибудь неясного стандартного значения. Это оказывается критически важным в Python, потому что имена заранее не объявляются – если бы Python предоставлял стандартные значения в случае использования в программе имен, которым не производилось присваивание, вместо того, чтобы трактовать их как ошибки, тогда было бы гораздо сложнее отлавливать опечатки в коде.
- Некоторые операции неявно выполняют присваивания. В текущем разделе мы сосредоточим внимание на операторе `=`, но в Python присваивания происходят во многих контекстах. Скажем, позже вы увидите, что импортирование модулей, определения функций и классов, переменные циклов `for` и аргументы функций являются неявными присваиваниями. Поскольку присваивание работает одинаково везде, где встречается, во время выполнения все упомянутые контексты просто связывают (т.е. присваивают) имена со ссылками на объекты.

Формы оператора присваивания

Несмотря на то что присваивание в Python является универсальной и всепроникающей концепцией, в главе нас интересует главным образом *операторы* присваивания. В табл. 11.1 проиллюстрированы различные формы оператора присваивания в Python вместе с их синтаксическими шаблонами.

Таблица 11.1. Формы оператора присваивания

Операция	Описание
<code>spam = 'Spam'</code>	Базовая форма
<code>spam, ham = 'yum', 'YUM'</code>	Присваивание кортежа (позиционное)
<code>[spam, ham] = ['yum', 'YUM']</code>	Присваивание списка (позиционное)
<code>a, b, c, d = 'spam'</code>	Присваивание последовательности, обобщенное
<code>a, *b = 'spam'</code>	Расширенная распаковка последовательности (Python 3.X)
<code>spam = ham = 'lunch'</code>	Групповое присваивание
<code>spams += 42</code>	Дополненное присваивание (эквивалентно <code>spams = spams + 42</code>)

Наиболее распространной безоговорочно считается первая форма присваивания в табл. 11.1: связывание имени (или компонента структуры данных) с одиночным объектом. В действительности вы могли бы сделать всю свою работу с помощью только этой базовой формы. В остальных строках табл. 11.1 представлены специальные формы, которые необязательны, но программисты часто находят их удобными на практике.

Распаковывающие присваивания кортежей и списков

Вторая и третья формы в табл. 11.1 являются родственными. Когда вы записываете кортеж или список слева от `=`, Python попарно соединяет объекты справа с целями слева по позициям, выполняя присваивание слева направо. Например, во второй строке табл. 11.1 имени `spam` присваивается строка `'yum'`, а имя `ham` связывается со строкой `'YUM'`. В данном случае Python внутренне может создавать кортеж элементов справа, поэтому такое присваивание называют распаковывающим.

Присваивания последовательностей

В недавних версиях Python присваивания кортежей и списков были обобщены в случае того, что теперь называется *присваиванием последовательности* — любой последовательности имен можно присваивать любую последовательность значений, и Python будет присваивать элементы позиционно по одному за раз. Мы можем даже смешивать типы задействованных последовательностей. Скажем, в четвертой строке табл. 11.1 кортеж имен попарно соединяется со строкой символов: а присваивается 's', б — 'р' и т.д.

Расширенная распаковка последовательности

Новая форма присваивания последовательности, доступная (только) в Python 3.X, обеспечивает большую гибкость в том, как мы выбираем части последовательности для присваивания. Например, в пятой строке табл. 11.1 имя *a* сопоставляется с первым символом строки справа, а имя *b* — с остатком строки: а присваивается 's', а *b* — ['р', 'а', 'м']. Мы имеем более простую альтернативу присваиванию результатов ручных операций нарезания.

Групповые присваивания

В шестой строке табл. 11.1 показана групповая форма присваивания. В групповой форме Python присваивает ссылку на один и тот же объект (крайний справа) всем целям слева. А табл. 11.1 именам *spam* и *ham* присваиваются ссылки на тот же самый строковый объект 'lunch'. Результат будет таким же, как если бы мы записали *ham = 'lunch'* и затем *spam = ham*, т.к. *ham* оценивается в исходный строковый объект (т.е. не отдельную копию этого объекта).

Дополненные присваивания

В последней строке табл. 11.1 приведен пример *дополненного присваивания* — сокращения, которое объединяет выражение и присваивание в лаконичную форму. Скажем, *spam += 42* дает такой же результат, как и *spam = spam + 42*, но дополненная форма требует меньшего набора и обычно выполняется быстрее. Вдобавок если объект является изменяемым и поддерживает эту операцию, тогда дополненное присваивание может выполняться даже еще быстрее за счет выбора операции обновления на месте взамен создания копии объекта. Как будет показано, для большинства бинарных операций выражений в Python существует один оператор дополненного присваивания.

Присваивание последовательности

В книге мы уже применяли и исследовали базовые присваивания, поэтому примем их как данность. Ниже демонстрируется несколько простых примеров распаковывающих присваиваний последовательностей в действии:

```
% python
>>> nudge = 1                      # Базовое присваивание
>>> wink = 2
>>> A, B = nudge, wink            # Присваивание кортежа
>>> A, B                         # Подобно A = nudge; B = wink
(1, 2)
>>> [C, D] = [nudge, wink]        # Присваивание списка
>>> C, D
(1, 2)
```

Обратите внимание, что в третьей строке взаимодействия мы на самом деле записали два кортежа, просто опустив заключающие круглые скобки. Python попарно соединяет значения в кортеже, указанном справа от операции присваивания, с переменными в кортеже слева и присваивает значения по одному за раз.

Присваивание кортежа приводит к распространенному кодовому трюку в Python, который был введен при решении упражнений в конце части II. Поскольку Python создает временный кортеж, который хранит исходные значения переменных справа, пока оператор выполняется, распаковывающие присваивания также представляют собой способ обмена значениями двух переменных без создания временной переменной — кортеж справа запоминает предыдущие значения переменных автоматически:

```
>>> nudge = 1
>>> wink = 2
>>> nudge, wink = wink, nudge # Кортежи: обмен значениями
>>> nudge, wink           # Подобно T = nudge; nudge = wink; wink = T
(2, 1)
```

Фактически первоначальные формы присваивания кортежей и списков в Python были обобщены для приема любого вида последовательности (в действительности итерируемого объекта) справа до тех пор, пока она имеет такую же длину, как последовательность слева. Можно присваивать кортеж значений списку переменных, строку символов кортежу переменных и т.д. Во всех случаях Python присваивает элементы в последовательности справа переменным в последовательности слева позиционно слева направо:

```
>>> [a, b, c] = (1, 2, 3)      # Присваивание кортежа значений списку имен
>>> a, c
(1, 3)
>>> (a, b, c) = "ABC"        # Присваивание строки символов кортежу
>>> a, c
('A', 'C')
```

Говоря формально, присваивание последовательности на самом деле поддерживает справа не только любую последовательность, но любой *итерируемый* объект. Он представляет собой более универсальную категорию, включающую коллекции как физические (скажем, списки), так и виртуальные (например, строки файла), которые были кратко определены в главе 4 и время от времени появлялись позже. Мы закрепим данный термин в главах 14 и 20, когда будем исследовать итерируемые объекты.

Расширенные шаблоны присваивания последовательностей

Хотя допускается смешивать типы последовательностей с обеих сторон символа `=`, мы обычно должны иметь *столько же* элементов справа, сколько имеем переменных слева, иначе возникнет ошибка. Python 3.X делает возможной более общую форму с расширенным синтаксисом распаковки `*`, который описан в следующем разделе. Но при нормальных обстоятельствах в Python 3.X — и всегда в Python 2.X — количества элементов в цели и в объекте присваивания обязаны совпадать:

```
>>> string = 'SPAM'
>>> a, b, c, d = string      # Одинаковые количества с обеих сторон
>>> a, d
('S', 'M')

>>> a, b, c = string        # В противном случае ошибка
...текст сообщения об ошибке не показан...
ValueError: too many values to unpack (expected 3)
Ошибка значения: слишком много значений для распаковки (ожидалось 3)
```

Для большей гибкости мы можем выполнять срезы в Python 2.X и 3.X. Задействовать нарезание для того, чтобы предыдущий случай заработал, можно разными способами:

```
>>> a, b, c = string[0], string[1], string[2:]      # Индексация и нарезание
>>> a, b, c
('S', 'P', 'AM')

>>> a, b, c = list(string[:2]) + [string[2:]]      # Нарезание и конкатенация
>>> a, b, c
('S', 'P', 'AM')

>>> a, b = string[:2]                                # То же самое, но проще
>>> c = string[2:]
>>> a, b, c
('S', 'P', 'AM')

>>> (a, b), c = string[:2], string[2:]            # Вложенные последовательности
>>> a, b, c
('S', 'P', 'AM')
```

Как демонстрирует последний пример во взаимодействии выше, мы даже можем присваивать *вложенные* последовательности, и Python ожидаемым образом распакует их части согласно их формам. В этом случае мы присваиваем кортеж из двух элементов, где первый элемент является вложенной последовательностью (строкой), в частности, как если бы мы записали его в следующем виде:

```
>>> ((a, b), c) = ('SP', 'AM')      # Попарно соединяются по форме и по позиции
>>> a, b, c
('S', 'P', 'AM')
```

Python попарно соединяет первую строку справа ('SP') с первым кортежем слева ((a, b)) и присваивает по одному символу за раз до присваивания целой второй строки ('AM') переменной с одновременно. В таком случае форма вложенной последовательности объекта слева должна совпадать с формой объекта справа. Присваивание вложенной последовательности подобного рода встречается редко, но оно может оказаться удобным для выборки частей структур данных с известными формами.

Скажем, в главе 13 мы увидим, что такая методика также работает в циклах `for`, потому что элементы цикла присваиваются цели, указанной в заголовке цикла:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ...      # Простое присваивание кортежей
for ((a, b), c) in [(1, 2), 3], ((4, 5), 6)]: ...    # Присваивание вложенных
                                                    # кортежей
```

В первой врезке “На заметку!” в главе 18 мы узнаем, что эта форма распаковывающего присваивания вложенного кортежа (на самом деле последовательности) работает и для списков аргументов функций в Python 2.X (хотя не в Python 3.X), т.к. аргументы функций тоже передаются путем присваивания:

```
def f(((a, b), c)): ... # Работает для аргументов в Python 2.X, но не в Python 3.X
f((1, 2), 3)
```

Распаковывающие присваивания последовательностей также дают начало еще одной распространенной кодовой идиоме в Python – присваивание серии целых чисел набору переменных:

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

Здесь инициализируются три имени целочисленными кодами 0, 1 и 2 соответственно (эквивалент Python *не* перечислимых типов данных, которые вы могли встречать в других языках). Чтобы понять такое присваивание, вам необходимо знать, что встроенная функция `range` генерирует список следующих друг за другом целых чисел (для отображения сразу всех их значений в Python 3.X потребуется вызов `list`):

```
>>> list(range(3))      # В Python 3.X требуется list()
[0, 1, 2]
```

Вызов `list` кратко затрагивался в главе 4; из-за частого использования функции `range` в циклах `for` мы подробно обсудим ее в главе 13.

Другим местом, где вы можете увидеть присваивание кортежей в работе, будет разбиение последовательности на ее голову и остальное в циклах, как иллюстрируется ниже:

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, L = L[0], L[1:]          # В следующем разделе описана
...     print(front, L)                # альтернатива * из Python 3.X
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Присваивание кортежей в цикле взамен можно было бы записать следующим образом, но зачастую удобнее объединить их вместе:

```
... front = L[0]
... L = L[1:]
```

Обратите внимание, что в этом коде список применяется как своего рода структура данных типа стека, которую нередко можно реализовать с помощью методов `append` и `pop` списковых объектов. Оператор `front = L.pop(0)` имел бы почти тот же эффект, что и оператор присваивания кортежа, но он оказался бы изменением на месте. В главе 13 вы узнаете гораздо больше о циклах `while` и других (часто лучших) способах прохода по последовательности посредством циклов `for`.

Расширенная распаковка последовательностей в Python 3.X

В предыдущем разделе демонстрировалось, как использовать ручное нарезание, чтобы сделать присваивания последовательностей более универсальными. С целью облегчения этого в Python 3.X (но не в Python 2.X) присваивание последовательности было обобщено. Если кратко, то в цели присваивания можно применять одиночное помеченное звездочкой имя, `*X`, для указания более универсального сопоставления с последовательностью – имени со звездочкой присваивается список, в котором собраны все элементы, не присвоенные остальным именам. Прием особенно удобен в распространенных кодовых шаблонах, таких как разбиение последовательности на ее “голову” и “остальное”, как в последнем примере предыдущего раздела.

Расширенная распаковка в действии

Давайте рассмотрим пример. Как мы уже видели, присваивания последовательностей по обыкновению требуют точно столько имен в цели слева, сколько есть элементов в объекте справа. Если длины не совпадают, тогда мы получим ошибку и в Python 2.X, и в Python 3.X (при условии, что мы не произведем вручную срез, как было показано в предыдущем разделе):

```
C:\code> c:\python33\python
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> print(a, b, c, d)
1 2 3 4

>>> a, b = seq
ValueError: too many values to unpack (expected 2)
Ошибка значения: слишком много значений для распаковки (ожидалось 2)
```

Однако в Python 3.X мы можем использовать в цели одиночное помеченнное звездочкой имя, чтобы обеспечить более универсальное сопоставление. В следующем продолжении нашего интерактивного сеанса `a` соответствует первому элементу в последовательности, `b` – все остальные:

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

В случае применения имени со звездочкой количество элементов в цели слева не обязано совпадать с длиной последовательности в объекте справа. На самом деле помеченнное звездочкой имя может появляться где угодно в цели. Скажем, в показанном далее взаимодействии `b` соответствует последнему элементу в последовательности, тогда как `a` – всему, что находится перед последним элементом:

```
>>> *a, b = seq
>>> a
[1, 2, 3]
>>> b
4
```

Когда имя со звездочкой располагается в середине, оно собирает все, что находится между другими указанными именами. Таким образом, в приведенном ниже взаимодействии `a` и `c` присваиваются первый и последний элементы, а `b` получает все, что между ними:

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

В более общем случае, где бы ни появлялось помеченнное звездочкой имя, ему будет присваиваться список, где собраны все элементы, которые не были присвоены именам в соответствующих позициях:

```
>>> a, b, *c = seq
>>> a
1
>>> b
2
>>> c
[3, 4]
```

Естественно, подобно нормальному присваиванию последовательности синтаксис расширенной распаковки работает для любого вида последовательности (в действительности любого *итерируемого* объекта), не только для списков. Вот распаковка символов в строки и чисел в результате вызова `range` (итерируемый объект в Python 3.X):

```
>>> a, *b = 'spam'  
>>> a, b  
('s', ['p', 'a', 'm'])  
>>> a, *b, c = 'spam'  
>>> a, b, c  
('s', ['p', 'a'], 'm')  
>>> a, *b, c = range(4)  
>>> a, b, c  
(0, [1, 2], 3)
```

Прием напоминает нарезание, но не является им в точности – распаковывающее присваивание последовательности всегда возвращает *список* для множества совпадающих элементов, тогда как нарезание возвращает последовательность того же самого типа, что и у нарезаемого объекта:

```
>>> s = 'spam'  
>>> s[0], s[1:]    # Срезы специфичны к типу, присваивание *  
                   # всегда возвращает список  
('s', 'pam')  
>>> s[0], s[1:3], s[3]  
('s', 'pa', 'm')
```

Учитывая такое расширение в Python 3.X для обработки списка, последний пример из предыдущего раздела становится еще проще, т.к. нам не приходится вручную нарезать, чтобы получить первый и остальные элементы:

```
>>> L = [1, 2, 3, 4]  
>>> while L:  
...     front, *L = L # Получение первого и остальных элементов без нарезания  
...     print(front, L)  
...  
1 [2, 3, 4]  
2 [3, 4]  
3 [4]  
4 []
```

Границные случаи

Несмотря на гибкость расширенной распаковки, полезно знать о некоторых граничных случаях. Во-первых, имя со звездочкой может соответствовать только одному символу, но этому имени всегда присваивается список:

```
>>> seq = [1, 2, 3, 4]  
>>> a, b, c, *d = seq  
>>> print(a, b, c, d)  
1 2 3 [4]
```

Во-вторых, если нет ничего, что можно было бы сопоставить с помеченным звездочкой именем, тогда ему присваивается пустой список независимо от того, где оно находится. В следующем коде `a`, `b`, `c` и `d` дали соответствия со всеми элементами в

последовательности, но Python присваивает е пустой список вместо трактовки случая как ошибочного:

```
>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

Наконец, в-третьих, ошибки по-прежнему могут возникать, если имен со звездочкой более одного, если обычных имен слишком мало и нет имени, помеченного звездочкой, и если имя со звездочкой само не записано внутри последовательности:

```
>>> a, *b, c, *d = seq
SyntaxError: two starred expressions in assignment
Ошибка синтаксиса: два имени со звездочкой в присваивании
>>> a, b = seq
ValueError: too many values to unpack (expected 2)
Ошибка значения: слишком много значений для распаковки (ожидалось 2)

>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple
Ошибка синтаксиса: имя со звездочкой должно находиться в списке или кортеже
>>> *a, = seq
>>> a
[1, 2, 3, 4]
```

Полезное удобство

Имейте в виду, что присваивание последовательности с расширенной распаковкой — это всего лишь удобство. Мы обычно в состоянии достичь тех же результатов с помощью явной индексации и нарезания (и фактически обязаны поступать так в Python 2.X), но расширенную распаковку проще записывать. Например, распространенный кодовый шаблон разбиения “первый, остальные” может быть записан любым из двух способов, но с нарезанием связана дополнительная работа:

```
>>> seq
[1, 2, 3, 4]
>>> a, *b = seq          # Первый, остальные
>>> a, b
(1, [2, 3, 4])
>>> a, b = seq[0], seq[1:]    # Первый, остальные: традиционный способ
>>> a, b
(1, [2, 3, 4])
```

Также часто встречающийся шаблон разбиения “остальные, последний” аналогично может быть записан любым из двух способов, но новый синтаксис расширенной распаковки требует заметно меньшего объема набора:

```
>>> *a, b = seq          # Остальные, последний
>>> a, b
([1, 2, 3], 4)
>>> a, b = seq[:-1], seq[-1]  # Остальные, последний: традиционный способ
>>> a, b
([1, 2, 3], 4)
```

Так как синтаксис расширенной распаковки последовательностей не только проще, но возможно естественнее, вполне вероятно, что со временем он станет более широко распространенным в коде Python.

Употребление в циклах `for`

Поскольку переменной цикла в операторе цикла `for` может быть любая цель присваивания, присваивание последовательности с расширенной распаковкой здесь также работает. Мы кратко упоминали инструмент итерации в форме цикла `for` в главе 4 и формально представим его в главе 13. В Python 3.X присваивание с расширенной распаковкой может появляться после слова `for`, где чаще всего используется простое имя переменной:

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
    ...
```

При использовании в таком контексте на каждой итерации Python просто присваивает кортежу имен очередной кортеж значений. Например, первая итерация выглядит так, как если бы мы выполнили следующий оператор присваивания:

```
a, *b, c = (1, 2, 3, 4) # b получает [2, 3]
```

Имена `a`, `b` и `c` можно применять внутри кода цикла для ссылки на извлеченные компоненты. По правде говоря, это вообще не специальный случай, а просто пример универсального присваивания в работе. Как было показано ранее в главе, то же самое мы можем делать посредством простого присваивания кортежа в Python 2.X и 3.X:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: # a, b, c = (1, 2, 3), ...
```

И мы всегда можем эмулировать поведение присваивания с расширенной распаковкой Python 3.X в Python 2.X за счет ручного нарезания:

```
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
    a, b, c = all[0], all[1:3], all[3]
```

Мы вернемся к этой теме в главе 13 при рассмотрении синтаксиса циклов `for`.

Групповые присваивания

Групповое присваивание просто присваивает всем заданным именам объект, указанный справа. Скажем, в следующем примере трем переменным `a`, `b` и `c` присваивается строка `'spam'`:

```
>>> a = b = c = 'spam'  
>>> a, b, c  
('spam', 'spam', 'spam')
```

Такая форма эквивалентна трем присваиваниям (но легче для записи):

```
>>> c = 'spam'  
>>> b = c  
>>> a = b
```

Групповое присваивание и разделяемые ссылки

Имейте в виду, что здесь есть всего лишь один объект, разделяемый тремя переменными (все они в итоге указывают на тот же самый объект в памяти). Такое поведение хорошо подходит для неизменяемых типов — например, при инициализации нулем набора счетчиков (вспомните, что переменным в Python должны быть присвоены зна-

чения, прежде чем их можно будет использовать, поэтому вы обязаны инициализировать счетчики нулем до того, как сможете их увеличивать):

```
>>> a = b = 0
>>> b = b + 1
>>> a, b
(0, 1)
```

Здесь модификация `b` изменяет только `b`, т.к. числа не поддерживают изменения на месте. Если присваиваемый объект является неизменяемым, то не имеет значения, сколько имен на него ссылаются.

Тем не менее, как всегда, мы должны быть более осторожными в случае инициализации переменных пустым изменяемым объектом вроде списка или словаря:

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

На этот раз, поскольку `a` и `b` ссылаются на тот же самый объект, его дополнение на месте через `b` также повлияет на то, что мы видим через `a`. В действительности мы имеем просто еще один пример явления разделяемых ссылок, с которым впервые столкнулись в главе 6. Во избежание проблемы инициализируйте изменяемые объекты в отдельных операторах, чтобы каждый из них создавал индивидуальный пустой объект, выполняя собственное литеральное выражение:

```
>>> a = []
>>> b = []          # a и b не разделяют тот же самый объект
>>> b.append(42)
>>> a, b
([], [42])
```

Присваивание кортежа следующего вида дает такой же результат – за счет выполнения двух списковых выражений оно создает два отдельных объекта:

```
>>> a, b = [], []
      # a и b не разделяют тот же самый объект
```

Дополненные присваивания

Начиная с версии Python 2.0, стал доступным набор дополнительных форматов операторов присваивания, который приведен в табл. 11.2. Известные как дополненные присваивания и позаимствованные из языка C, такие форматы по большей части являются всего лишь сокращениями. Они заключают в себе бинарную операцию и присваивание. Скажем, следующие два формата практически равнозначны:

```
X = X + Y           # Традиционная форма
X += Y              # Более новая дополненная форма
```

Таблица 11.2. Операторы дополненного присваивания

X += Y	X &= Y	X -= Y	X = Y
X *= Y	X ^= Y	X /= Y	X >>= Y
X %= Y	X <<= Y	X **= Y	X //= Y

Дополненное присваивание работает на любом типе, который поддерживает подразумеваемую бинарную операцию. Например, вот два способа добавления 1 к имени:

```
>>> x = 1
>>> x = x + 1      # Традиционное присваивание
>>> x
2
>>> x += 1         # Дополненное присваивание
>>> x
3
```

Когда дополненная форма применяется к последовательности, подобной строке, она взамен выполняет конкатенацию. Таким образом, вторая строка здесь эквивалентна более длинному оператору $S = S + "SPAM"$:

```
>>> S = "spam"
>>> S += "SPAM"      # Подразумевается конкатенация
>>> S
'spamSPAM'
```

Как показано в табл. 11.2, существуют аналогичные дополненные формы присваивания для большинства бинарных операций выражений Python (т.е. операций со значениями в левой и правой сторонах). Скажем, $X *= Y$ умножает и присваивает, $X >= Y$ сдвигает вправо и присваивает и т.д. Деление с округлением в меньшую сторону ($X // Y$) было добавлено в версии Python 2.2.

Дополненные присваивания обладают тремя преимуществами¹.

- Уменьшается объем набора на клавиатуре. Нужно ли продолжать?
- Левая сторона должна оцениваться только раз. В $X += Y$ имя X может быть сложным выражением с объектами. В дополненной форме его код потребуется выполнить лишь однократно. Однако в длинной форме, $X = X + Y$, имя X встречается два раза и должно быть оценено дважды. По этой причине дополненные присваивания обычно выполняются быстрее.
- Оптимальная методика выбирается автоматически. То есть для объектов, которые поддерживают изменения на месте, дополненные формы автоматически выполняют операции изменения на месте взамен более медленного создания их копий.

Последний пункт требует чуть больших пояснений. В случае дополненных присваиваний в качестве оптимизации к изменяемым объектам могут применяться операции изменения на месте. Вспомните, что списки допускают расширение разнообразными путями. Чтобы добавить одиничный элемент в конец списка, мы можем использовать конкатенацию или вызов `append`:

```
>>> L = [1, 2]
>>> L = L + [3]      # Конкатенация: медленнее
>>> L
[1, 2, 3]
>>> L.append(4)     # Быстрее, но на месте
>>> L
[1, 2, 3, 4]
```

¹ Программисты на C/C++ заметят, что хотя Python теперь поддерживает операторы вроде $X += Y$, он все еще не располагает операциями инкремента/декремента (например, $X++$, $--X$). Они не вполне укладываются в объектную модель Python, потому что в Python отсутствует понятие изменений *на месте* для неизменяемых объектов, таких как числа.

И для добавления набора элементов в конец списка мы можем либо снова выполнить конкатенацию, либо вызвать списковый метод `extend`²:

```
>>> L = L + [5, 6]          # Конкатенация: медленнее
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])      # Быстрее, но на месте
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

В обоих случаях конкатенация менее подвержена побочным эффектам, связанным с разделяемыми ссылками на объекты, но в целом будет выполняться медленнее, чем эквивалентное изменение на месте. Операция конкатенации обязана создать новый объект, скопировать список слева и затем скопировать список справа. Напротив, вызов метода изменения на месте просто добавляет элементы в конец блока памяти (внутренне он может быть устроен сложнее, но такого описания вполне достаточно).

Когда для расширения списка используется дополненное присваивание, мы почти полностью можем забыть об этих деталях — Python автоматически вызывает более быстрый метод `extend` вместо применения медленной операции конкатенации, подразумеваемой `+=`:

```
>>> L += [9, 10]           # Отображается на L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Тем не менее, следует отметить, что из-за такой эквивалентности присваивание `+=` для списка не точно соответствует операциям `+` и `=` во всех случаях — для списков присваивание `+=` допускает произвольные последовательности (как и метод `extend`), но конкатенация обычно нет:

```
>>> L = []
>>> L += 'spam'    # += и extend допускают любую последовательность, но + нет!
>>> L
['s', 'p', 'a', 'm']
>>> L = L + 'spam'
TypeError: can only concatenate list (not "str") to list
Ошибка типа: выполнять конкатенацию можно только списка (не строки) со списком
```

Дополненное присваивание и разделяемые ссылки

Как правило, описанное поведение является желательным, но обратите внимание: отсюда вытекает, что присваивание `+=` представляет собой изменение *на месте* для списков; таким образом, оно не в точности похоже на конкатенацию `+`, которая всегда создает *новый* объект. Как и для всех случаев с разделяемыми ссылками, это отличие может иметь значение, когда на изменяемый объект ссылаются другие имена:

```
>>> L = [1, 2]
>>> M = L                  # L и M ссылаются на тот же самый объект
>>> L = L + [3, 4]          # Конкатенация создает новый объект
>>> L, M                  # Изменяет L, но не M
([1, 2, 3, 4], [1, 2])
```

² Как предполагалось в главе 6, мы также можем использовать присваивание по срезу (например, `L[len(L):] = [11, 12, 13]`), но оно работает практически аналогично более простому и понятному списковому методу `extend`.

```

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4]           # Но += действительно означает extend
                           # M тоже видит изменение на месте!
>>> L, M
([1, 2, 3, 4], [1, 2, 3, 4])

```

Это имеет значение только для изменяемых объектов вроде списков и словарей, а сам сценарий малоизвестен (во всяком случае, пока он не возникнет в вашем коде!). Как всегда, создавайте копии своих изменяемых объектов, если хотите разрушить структуру с разделяемыми ссылками.

Правила именования переменных

После исследования операторов присваивания наступило время обсудить более формально использование имен переменных. В Python имена начинают свое существование, когда им присваиваются значения, но есть несколько правил, которым необходимо следовать при выборе имен для объектов программы.

Синтаксис: (*подчеркивание или буква*) + (*любое количество букв, цифр или подчеркиваний*)

Имена переменных должны начинаться с подчеркивания или буквы, за которой может следовать любое количество букв, цифр или подчеркиваний. Например, `_spam`, `spam` и `Spam_1` – допустимые имена, но `1_Spam`, `spam$` и `@#!` – нет.

Регистр символов имеет значение: `SPAM` – не то же самое, что и `spam`

Python всегда обращает внимание на регистр символов в программах, как в создаваемых вами именах, так и в зарезервированных словах. Скажем, имена `X` и `x` относятся к двум разным переменным. Для обеспечения переносимости регистр символов также имеет значение в именах импортируемых файлов модулей даже на платформах, где файловые системы нечувствительны к регистру. В результате импортирование продолжает нормально работать после копирования программ на другие платформы.

Зарезервированные слова не разрешены

Определяемые вами имена не могут совпадать со словами, которые означают особые вещи в языке Python. Например, если вы попытаетесь выбрать для переменной имя `class`, то Python сообщит об ошибке, но имена `klass` и `Class` будут допустимыми. В табл. 11.3 перечислены слова, которые в текущий момент являются зарезервированными в Python (и потому не разрешены для применения для имен переменных).

Таблица 11.3. Зарезервированные слова Python 3.X

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

Зарезервированные слова в табл. 11.3 специфичны для Python 3.X. В Python 2.X набор зарезервированных слов слегка отличается:

- `print` – зарезервированное слово, потому что `print` является оператором, а не встроенной функцией (о чем пойдет речь позже в главе);
- `exec` – зарезервированное слово, т.к. `exec` является оператором, а не встроенной функцией;
- `nonlocal` – не зарезервированное слово, поскольку такой оператор отсутствует.

В более старых версиях Python положение дел более-менее такое же, но с несколькими вариациями:

- `with` и `as` не были зарезервированными словами до версии Python 2.6, когда официально стали доступными диспетчеры контекстов;
- `yield` не было зарезервированным словом до версии Python 2.3, когда появились генераторные функции;
- `yield` трансформировалось из оператора в выражение в версии Python 2.5, но по-прежнему является зарезервированным словом, а не встроенной функцией.

Как видите, большинство зарезервированных слов Python записано в нижнем регистре символов. Кроме того, они все по-настоящему зарезервированы – в отличие от имен во встроенной области видимости, которые встречаются в следующей части книги, вы не можете переопределять зарезервированные слова путем присваивания (например, `and = 1` приводит к синтаксической ошибке)³.

Помимо представления в смешанном регистре символов первые три элемента в табл. 11.3, `True`, `False` и `None`, имеют несколько необычный смысл – они также обнаруживаются во встроенной области видимости, описанной в главе 17, и формально являются именами, которым присваиваются объекты. Однако в Python 3.X указанные зарезервированные слова действительно зарезервированы во всех отношениях и не могут использоваться для каких-либо других целей в сценарии кроме объектов, которые они представляют. Все остальные зарезервированные слова жестко вшиты в синтаксис Python и могут появляться только в специфических контекстах, для которых они предназначены.

Более того, поскольку имена модулей в операторах `import` становятся переменными в ваших сценариях, ограничения имен переменных также распространяются на имена файлов модулей. Скажем, вы можете иметь файлы кода с именами `and.py` и `my-code.py` и запускать их как сценарии верхнего уровня, но не можете импортировать их: имена файлов без расширения `.py` становятся переменными в коде, а потому обязаны следовать только что обрисованным правилам именования переменных. Зарезервированные слова не разрешены, так что символы – не допускаются, хотя подчеркивания применять можно. Мы вернемся к этой концепции, связанной с модулями, в части V книги.

³ Во всяком случае, в стандартном CPython. Альтернативные реализации Python могут разрешать именам переменных совпадать с зарезервированными словами Python. Обзор альтернативных реализаций, таких как `Jython`, приводился в главе 2.

Протокол объявления функциональности нежелательной к использованию в Python

Интересно обратить внимание на то, как изменения в зарезервированных словах постепенно и поэтапно вносятся в язык. Когда новая функциональная возможность способна нарушить работу существующего кода, Python обычно делает ее необязательной и начинает выдавать предупреждения о “нежелательной к использованию” функциональности в одном или большем числе выпусков, прежде чем возможность станет официально доступной. Идея заключается в том, что вы должны располагать достаточным временем, чтобы заметить предупреждения и обновить свой код до перехода на новый выпуск. Сказанное не относится к крупным новым выпускам вроде Python 3.0 (который неограниченно нарушил работу существующего кода), но в целом верно в остальных случаях.

Например, `yield` было необязательным расширением в Python 2.2, но начиная с Python 2.3, является стандартным ключевым словом. Оно применяется в сочетании с генераторными функциями. Это было одним из немногих случаев, когда в Python нарушалась обратная совместимость. Тем не менее, ключевое слово `yield` вводилось поэтапно на протяжении длительного времени: в версии Python 2.2 начали выдаватьсь предупреждения о функциональности, нежелательной к использованию, и оно не было задействовано вплоть до версии Python 2.3. Подобным же образом в Python 2.6 слова `with` и `as` становятся новыми зарезервированными словами для применения в диспетчерах контекста (более новая форма обработки исключений). Указанные два слова не являются зарезервированными в Python 2.5, если только вручную не включить средство диспетчеров контекста с помощью `from __future__ import` (обсуждается позже в книге). При использовании в Python 2.5 слова `with` и `as` вызывают генерацию предупреждений о предстоящем изменении. Исключением будет версия IDLE в Python 2.5, которая делает данное средство доступным (т.е. применение слов `with` и `as` для имен переменных приводит к ошибкам в Python 2.5, но только в версии с графическим пользовательским интерфейсом IDLE).

Соглашения по именованию

Кроме описанных выше правил существует также набор соглашений по именованию – правил, которые не считаются обязательными, но соблюдаются в повседневной практике. Скажем, из-за того, что имена с двумя подчеркиваниями в начале и конце (например, `__name__`) обычно имеют особый смысл для интерпретатора Python, вы должны избегать использования такого шаблона в собственных именах. Ниже приведен перечень соглашений по именованию, принятых в Python.

- Имена, которые начинаются с одиночного подчеркивания (`_X`), не импортируются оператором `from module import *` (описанным в главе 23).
- Имена с двумя подчеркиваниями в начале и конце (`__X__`) являются системными именами, которые имеют особый смысл для интерпретатора.
- Имена, начинающиеся с двух подчеркиваний, но не оканчивающиеся ими (`__X`), локализованы (“искажены”) для включения в себя классов (см. обсуждение псевдооткрытых атрибутов в главе 31).
- Имя, состоящее из одиночного подчеркивания (`_`), хранит результат последнего выражения при работе в интерактивном сеансе.

В дополнение к перечисленным соглашениям, связанным с интерпретатором Python, существует ряд других соглашений, которым по обыкновению следуют про-

граммисты на Python. Например, позже в книге вы увидите, что имена классов, как правило, начинаются с буквы верхнего регистра, а имена модулей — с буквы нижнего регистра, и что имя `self`, не будучи зарезервированным, обычно исполняет специальную роль в классах. В главе 17 также будет исследоваться еще одна более широкая категория имен, известных как *встроенные*, которые предварительно определены, но не зарезервированы (и потому допускают повторное присваивание: `open = 42` работает, хотя временами вам захочется, чтобы это было не так!).

Имена не имеют тип, но объекты имеют

В главе по большей части приводится обзор, но помните, что крайне важно четко различать имена и объекты в Python. Как было описано в главе 6, объекты имеют тип (скажем, целое число, список) и могут быть изменяемыми или нет. С другой стороны, имена (они же переменные) всегда являются лишь ссылками на объекты; с ними не связаны понятие изменяемости и информация о типе, не считая типа объекта, на который они ссылаются в заданный момент времени.

Таким образом, в разные моменты времени одному и тому же имени можно присваивать разные виды объектов:

```
>>> x = 0          # x связывается с объектом целого числа
>>> x = "Hello"    # Теперь x - строка
>>> x = [1, 2, 3]  # А теперь x - список
```

В более поздних примерах вы увидите, что такая обобщенная природа имен может оказываться решающим преимуществом в программировании на Python. В главе 17 вы узнаете, что имена существуют в так называемой *области видимости*, которая определяет то, где они могут применяться; место присваивания имени устанавливает то, где оно будет видимым⁴.



Дополнительные предложения относительно именования можно найти в обсуждении соглашений по именованию, которое предлагается в полуофициальном руководстве по стилю Python, называемом *PEP 8*. Руководство доступно по ссылке <http://www.python.org/dev/peps/pep-0008>. Формально данный документ систематизирует стандарты кодирования для библиотечного кода Python.

Несмотря на полезность стандартов кодирования, здесь применимы обычные предостережения. Во-первых, PEP 8 предлагает слишком тонкие детали, которые вы еще вероятно не готовы воспринимать к настоящему моменту. Во-вторых, откровенно говоря, он стал более сложным, жестким и субъективным, чем могло быть необходимо — некоторые его предложения вообще не приняты повсеместно или не соблюдаются программистами на Python, выполняющими реальную работу. Более того, в ряде самых известных компаний, использующих Python в наши дни, были приняты собственные стандарты кодирования, которые отличаются от PEP 8.

Вместе с тем руководство PEP 8 систематизирует полезные эмпирические знания о Python и великолепно подходит для новичков, если воспринимать приводимые в нем предложения в качестве рекомендаций, а не догмы.

⁴ Если вы использовали более ограничительный язык вроде C++, то вам небезинтересно будет узнать, что в Python отсутствует понятие объявления `const`, имеющееся в C++; определенные объекты могут быть *неизменяемыми*, но присваивать значения именам можно всегда. В Python также предусмотрены способы скрытия имен в классах и модулях, но они не похожи на объявления в C++ (если вас интересует скрытие атрибутов, тогда ознакомьтесь с описанием имен модулей `_X` в главе 25, имен классов `_X` в главе 31 (том 2), а также с примером декораторов классов `Private` и `Public` в главе 39 тома 2).

Операторы выражений

В Python выражение можно также применять как оператор, т.е. в отдельной строке. Но поскольку результат выражения не сохраняется, поступать так имеет смысл, только когда выражение делает что-то полезное в виде побочного эффекта. Выражения используются как операторы обычно в двух ситуациях.

Для вызова функций и методов

Некоторые функции и методы выполняют свою работу и не возвращают какое-либо значение. В других языках такие функции иногда называются *процедурами*. Из-за того, что они не возвращают значений, в сохранении которых вы могли бы быть заинтересованы, их допускается вызывать посредством операторов выражений.

Для вывода значений в интерактивной подсказке

Python отображает результаты выражений, набираемых в командной строке интерактивной подсказки. Формально они также являются операторами выражений; они служат сокращениями для набора операторов `print`.

В табл. 11.4 описаны распространенные формы операторов выражений в Python. Вызовы функций и методов записываются с нулем или большим количеством объектов аргументов (в действительности выражений, результатом оценки которых будут объекты) в круглых скобках после имени функции/метода.

Таблица 11.4. Распространенные операторы выражений Python

Операция	Описание
<code>spam(eggs, ham)</code>	Вызовы функций
<code>spam.ham(eggs)</code>	Вызовы методов
<code>spam</code>	Вывод переменных в интерактивном интерпретаторе
<code>print(a, b, c, sep='')</code>	Вывод результатов операций в Python 3.X
<code>yield x ** 2</code>	Выдача значений в операторах выражений

Последние две строки в табл. 11.4 представляют отчасти специальные случаи — как вы увидите далее в главе, вывод в Python 3.X является вызовом функции, обычно записываемым в отдельной строке, а операция `yield` в генераторных функциях (рассматриваются в главе 20) также обычно записывается как оператор. Оба случая — на самом деле всего лишь примеры операторов выражений.

Скажем, хотя вы, как правило, выполняете вызов `print` из Python 3.X в отдельной строке как оператор выражения, он возвращает значение подобно вызову любой другой функции (его возвращаемым значением будет `None`, принятное по умолчанию для функций, которые не возвращают что-либо важное):

```
>>> x = print('spam')      # print в Python 3.X - выражение вызова функции
spam
>>> print(x)            # Но оно записывается как оператор выражения
None
```

Также имейте в виду, что хотя в Python выражения выглядеть как операторы, операторы не могут использоваться в качестве выражений. Оператор, который не является выражением, в общем случае обязан находиться в отдельной строке, а не вкладываться

в более крупную синтаксическую структуру. Например, Python не позволяет встраивать операторы присваивания (=) в другие выражения. Логическое обоснование такого решения связано с желанием избежать часто допускаемых ошибок при написании кода; вы не сможете непредумышленно изменить переменную, набрав =, когда на самом деле намеревались применить проверку на предмет равенства (==). Вы увидите, как обойти это ограничение, в главе 13 при обсуждении цикла while языка Python.

Операторы выражений и изменения на месте

В итоге мы подобрались к еще одной распространенной ошибке при работе с Python. Операторы выражений часто используются для выполнения списковых методов, которые изменяют список на месте:

```
>>> L = [1, 2]
>>> L.append(3)           # Добавление представляет собой изменение на месте
>>> L
[1, 2, 3]
```

Однако новички в Python нередко взамен записывают такое действие в виде оператора присваивания, намереваясь присвоить L список большего размера:

```
>>> L = L.append(4)      # Но append возвращает None, а не L
>>> print(L)            # Таким образом, мы утрачиваем наш список!
None
```

Тем не менее, прием совершенно не работает. Обращение к операции изменения на месте, такой как append, sort или reverse, на списке всегда модифицирует его на месте, но упомянутые методы возвращают не измененный список, а объект None. Соответственно, если вы присвоите имени переменной результат подобной операции, то фактически утратите список (и в процессе он, возможно, будет подвержен сборке мусора!).

Мораль этой истории – вызывать операции изменения на месте, не присваивая их результаты. Мы еще вернемся к данному явлению в разделе “Распространенные затруднения при написании кода” главы 15, потому что оно также встречается в контексте ряда операторов цикла, которые будут рассматриваться в будущих главах.

Операции вывода

В Python различные объекты выводятся с помощью print – простого дружественного к программисту интерфейса к стандартному потоку вывода.

Формально вывод преобразует один или большее количество объектов в их текстовые представления, добавляет незначительное форматирование и отправляет результатирующий текст либо в стандартный поток вывода, либо в другой поток данных, подобный файлу. Говоря чуть более подробно, print прочно связан с понятиями файлов и потоков данных в Python.

Методы файловых объектов

В главе 9 вы узнали о методах файловых объектов, которые записывают текст (например, file.write(str)). Операции вывода похожи, но более специализированы – в то время, как методы файловых объектов записывают строки в произвольные файлы, print по умолчанию записывает объекты в поток stdout, автоматически добавляя к ним некоторое форматирование. В отличие от методов файловых объектов в случае применения операций вывода преобразовывать объекты в строки не требуется.

Стандартный поток вывода

Стандартный поток вывода (часто известный как `stdout`) – это просто место, куда по умолчанию отправляется текстовый вывод программы. Наряду со стандартными потоками ввода и ошибок он является одним из трех подключений к данным, созданным при запуске сценария. Стандартный поток вывода обычно отображается на окно, где запущена программа Python, если только он не был перенаправлен в файл или конвейер в командной оболочке операционной системы.

Поскольку стандартный поток вывода доступен в Python как файловый объект `stdout` из встроенного модуля `sys` (т.е. `sys.stdout`), можно эмулировать `print` посредством обращения к методам записи файловых объектов. Однако `print` значительно проще использовать, а также легко выводить текст в другие файлы и потоки данных.

Вывод также представляет собой одно из самых заметных мест, где версии Python 3.X и Python 2.X расходятся. В действительности такое расхождение обычно является первой причиной, по которой большинство кода Python 2.X не будет работать в неизменном виде под управлением Python 3.X. В частности, способ написания операций вывода зависит от того, какая версия Python применяется:

- в Python 3.X операция вывода представляет собой встроенную функцию с ключевыми аргументами для специальных режимов;
- в Python 2.X операция вывода представляет собой оператор со своим специфическим синтаксисом.

Поскольку книга охватывает Python 3.X и 2.X, мы рассмотрим все формы по очереди. Если вам достаточно повезло, чтобы иметь дело с кодом, написанным только для одной версии Python, тогда можете читать подходящий раздел. Тем не менее, ситуация может измениться, поэтому не помешает ознакомиться с обоими случаями. Кроме того, пользователи последних выпусков Python 2.X при желании могут также импортировать и использовать разновидность вывода из Python 3.X – из-за его дополнительной функциональности и для облегчения будущей миграции в Python 3.X.

Функция `print` в Python 3.X

Строго говоря, в Python 3.X вывод не имеет форму отдельного оператора. Взамен он представляет собой пример *оператора выражения*, который мы исследовали в предыдущем разделе.

Встроенная функция `print` обычно вызывается в отдельной строке, т.к. она не возвращает значение, заслуживающее внимания (формально `print` возвращает `None`, как упоминалось в предыдущем разделе). Однако из-за того, что `print` является нормальной функцией, при выводе в Python 3.X применяется *стандартный синтаксис вызова функций*, а не какая-то специальная операторная форма. И поскольку предоставляются специальные рабочие режимы посредством ключевых аргументов, такая форма более универсальна и лучше поддерживает будущие усовершенствования.

По сравнению с этим оператор `print` в Python 2.X имеет специальный синтаксис для поддержки таких расширений, как подавление концов строки и перенаправление в файлы. Кроме того, оператор Python 2.X вообще не поддерживает указание разделителей; в Python 2.X вам чаще приходится создавать строки заранее, чем в Python 3.X. Вместо добавления дополнительного специального синтаксиса для оператора `print` в Python 3.X принят единственный универсальный подход, охватывающий все случаи.

Формат вызова

С точки зрения синтаксиса вызов функции `print` в Python 3.X имеет следующую форму (аргумент `flush` появился в версии Python 3.3):

```
print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout][, flush=False])
```

В показанной формальной записи элементы в квадратных скобках являются необязательными и в заданном вызове могут быть опущены, а значения после символа `=` предоставляют стандартные значения для аргументов. Выражаясь упрощенно,строенная функция `print` выводит в поток данных `file` текстовые представления одного или большего числа объектов `object`, разделенные строкой `sep`, за которыми следует строка `end`, сбрасывая или не сбрасывая буферизированный вывод согласно аргументу `flush`.

Части `sep`, `end`, `file` и (в Python 3.3 и последующих версиях) `flush`, когда присутствуют, должны указываться в виде *ключевых аргументов* – т.е. вы обязаны использовать специальный синтаксис `имя=значение` для передачи аргументов по имени, а не по позиции. Ключевые аргументы подробно обсуждаются в главе 18 и применять их легко. Ключевые аргументы, передаваемые вызову, могут указываться в любом порядке слева направо после объектов, подлежащих выводу, и они управляют работой `print`.

- `sep` – строка, вставляемая между текстовыми представлениями объектов, которой по умолчанию будет одиночный пробел, если она не задана; передача пустой строки полностью подавляет разделители.
- `end` – строка, добавляемая в конец выводимого текста, которой по умолчанию будет символ новой строки `\n`, если она не задана. Передача пустой строки позволяет избежать перехода на следующую строку вывода в конце выводимого текста – следующий вызов `print` продолжит добавление с конца текущей строки вывода.
- `file` указывает файл, стандартный поток данных или другой объект, подобный файлу, в который будет отправляться текст; если он не передан, то по умолчанию принимается поток стандартного вывода `sys.stdout`. Передавать можно любой объект, имеющий метод `write(string)`, как у файловых объектов, но реальные файлы должны быть уже открыты для вывода.
- `flush`, появившийся в Python 3.3, по умолчанию принимается равным `False`. Он позволяет вызову `print` предписывать, что текст должен немедленно сбрасываться через поток вывода в любые ожидающие получатели. Обычно буферизация в памяти вывода определяется аргументом `file`; передача в аргументе `flush` значения `True` принудительно сбрасывает поток данных.

Текстовое представление каждого выводимого объекта `object` получается путем передачи объекта встроенной функции `str` (или ее эквиваленту в Python); вы увидите, что эта встроенная функция возвращает “дружественную к пользователю” строку отображения для любого объекта⁵. Вызов функции `print` без аргументов просто помещает в поток стандартного вывода символ новой строки, который обычно отображается как пустая строка.

⁵ Формально `print` использует эквивалент `str` во внутренней реализации Python, но результат будет таким же. Помимо исполнения роли преобразования в строку `str` также является именем строкового типа данных и за счет указания дополнительного аргумента `encoding` может применяться для декодирования строк, как будет показано в главе 37; вторая роль относится к расширенному использованию `str`, так что здесь мы ее можем благополучно проигнорировать.

Функция print из Python 3.X в действии

Вывод в Python 3.X вероятно проще, чем может следовать из некоторых его деталей. В целях иллюстрации давайте рассмотрим несколько коротких примеров. Приведенные далее вызовы `print` выводят объекты разнообразных типов в поток стандартного вывода со стандартным разделителем и добавленным форматированием в виде символа конца строки (они приняты по умолчанию из-за того, что представляют самый распространенный сценарий использования):

```
C:\code> c:\python33\python
>>> print()                                # Отображение пустой строки
>>> x = 'spam'
>>> y = 99
>>> z = ['eggs']
>>>
>>> print(x, y, z)  # Вывод трех объектов со стандартными значениями аргументов
spam 99 ['eggs']
```

Здесь нет нужды преобразовывать объекты в строки, как требовалось бы для файловых методов записи. По умолчанию вызовы `print` добавляют пробел между выводимыми объектами. Чтобы подавить добавление пробела, укажите в ключевом аргументе `sep` либо пустую строку, либо задайте альтернативный разделитель по своему выбору:

```
>>> print(x, y, z, sep='')      # Подавить разделитель
spam99['eggs']
>>>
>>> print(x, y, z, sep=' ', ')  # Специальный разделитель
spam, 99, ['eggs']
```

Также `print` по умолчанию добавляет символ конца строки для завершения строки вывода. Вы можете подавить его и избежать разрыва строки, передав пустую строку в ключевом аргументе `end`, или указать другой символ завершения, включая `\n` для ручного разрыва строки (во второй строке ниже находятся два оператора, разделенные точкой с запятой):

```
>>> print(x, y, z, end='')      # Подавление разрывов строк
spam 99 ['eggs']>>>
>>>
>>> print(x, y, z, end=''); print(x, y, z)    # Два оператора print,
                                                 # выводящие в одну строку
spam 99 ['eggs']spam 99 ['eggs']
>>> print(x, y, z, end='...\n')                 # Специальный конец строки
spam 99 ['eggs']...
>>>
```

Вы можете также комбинировать ключевые аргументы, указывая строки разделителя и конца строки — они могут следовать в любом порядке, но находиться после всех выводимых объектов:

```
>>> print(x, y, z, sep='...', end='!\n')      # Множество ключевых аргументов
spam...99...['eggs']!
>>> print(x, y, z, end='!\n', sep='...')        # Порядок не имеет значения
spam...99...['eggs']!
```

Ниже показано, как применяется ключевой аргумент `file` — он направляет выводимый текст в открытый выходной файл или в другой совместимый объект на протяжении действия одиночного вызова `print` (на самом деле это форма перенаправления потока данных, к которой мы вскоре вернемся):

```

>>> print(x, y, z, sep='...', file=open('data.txt', 'w')) # Вывод в файл
>>> print(x, y, z)                                         # Возврат к stdout
spam 99 ['eggs']
>>> print(open('data.txt').read())      # Отображение текста файла
spam...99...['eggs']

```

Наконец, имейте в виду, что варианты разделителя и конца строки, предлагаемые операцией `print`, являются всего лишь удобством. Если вам необходимо отображать с более специфическим форматированием, тогда не выводите подобным образом. Взамен создавайте более сложную строку заранее или внутри самого вызова `print` с использованием строковых инструментов, которые обсуждались в главе 7, и выводите сразу всю строку:

```

>>> text = '%s: %-.4f, %05d' % ('Result', 3.14159, 42)
>>> print(text)
Result: 3.1416, 00042
>>> print('%s: %-.4f, %05d' % ('Result', 3.14159, 42))
Result: 3.1416, 00042

```

Как вы увидите в следующем разделе, почти все аспекты, касающиеся функции `print` из Python 3.X, также применимы напрямую к оператору `print` в Python 2.X; это имеет смысл, памятуя о том, что функция `print` предназначалась для эмуляции и усовершенствования поддержки вывода Python 2.X.

Оператор `print` в Python 2.X

Ранее уже упоминалось, что вывод в Python 2.X использует оператор с уникальным и специфическим синтаксисом, а не встроенную функцию. Тем не менее, с точки зрения практики вывод Python 2.X по большей части является вариацией на тему; за исключением строки разделителя (поддерживается в Python 3.X, но не в Python 2.X) и сброса вывода (доступного, начиная с версии Python 3.3) все, что мы можем делать с функцией `print` из Python 3.X, имеет прямой аналог в операторе `print` из Python 2.X.

Формы оператора

В табл. 11.5 перечислены формы оператора `print` в Python 2.X вместе с эквивалентными вызовами функции `print` из Python 3.X. Обратите внимание, что запятая в операторах `print` важна — она разделяет объекты, подлежащие выводу, а хвостовая запятая подавляет символ конца строки, которым обычно дополняется выведенный текст (не перепутайте с синтаксисом кортежей!). Здесь также применяется синтаксис `>>`, используемый для операции сдвига вправо, чтобы указать целевой поток вывода, который отличается от стандартного `sys.stdout`.

Таблица 11.5. Формы оператора `print` в Python 2.X

Оператор Python 2.X	Эквивалент Python 3.X	Описание
<code>print x, y</code>	<code>print(x, y)</code>	Выводит текстовые представления объектов в <code>sys.stdout</code> ; добавляет пробел между элементами и конец строки после текста
<code>print x, y,</code>	<code>print(x, y, end='')</code>	То же самое, но без добавления конца строки после текста
<code>print>>afile,x,y</code>	<code>print(x, y, file=afile)</code>	Отправляет текст методу <code>afile.write</code> , а не <code>sys.stdout.write</code>

Оператор print из Python 2.X в действии

Хотя оператор `print` в Python 2.X обладает более уникальным синтаксисом, чем функция `print` в Python 3.X, применять их одинаково легко. Давайте снова обратимся к примерам. Следующий оператор `print` в Python 2.X добавляет пробел между элементами, разделенными запятыми, и по умолчанию помещает разрыв строки в конце текущей строки вывода:

```
C:\code> c:\python27\python
>>> x = 'a'
>>> y = 'b'
>>> print x, y
a b
```

Такое форматирование просто принимается по умолчанию; вы сами можете решить, использовать его или нет. Для подавления разрыва строки, чтобы позже можно было добавить дополнительный текст в текущую строку вывода, поместите в конец оператора `print` запятую, как показано во второй строке табл. 11.5 (точка с запятой снова применяется для разделения двух операторов в одной строке):

```
>>> print x, y; print x, y
a b a b
```

Чтобы подавить пробелы между элементами, не выводите так. Взамен создайте выходную строку с использованием инструментов конкатенации и форматирования строк, описанных в главе 7, и выводите сразу всю строку:

```
>>> print x + y
ab
>>> print '%s...%s' % (x, y)
a...b
```

Как видите, кроме специального синтаксиса для режимов применения использовать операторы `print` в Python 2.X практически настолько же просто, как и функцию `print` из Python 3.X. В следующем разделе раскрывается способ указания файлов в операторах `print` из Python 2.X.

Перенаправление потока вывода

В Python 3.X и 2.X средство `print` по умолчанию отправляет текст в стандартный поток вывода. Однако часто удобно отправлять его куда-то в другое место — например, в текстовый файл с целью сохранения результатов для более поздней обработки или проведения тестирования. Хотя такое перенаправление можно выполнить в командной оболочке системы вне самого Python, оказывается, что перенаправлять потоки данных не менее легко и внутри самого сценария.

Программа "hello world" на Python

Начнем с обычного (и практически бессмысленного) эталонного теста языка — программы “hello world”. Чтобы отобразить сообщение “hello world” в Python, нужно просто вывести строку с помощью доступной версии операции `print`:

```
>>> print('hello world')      # Вывод строкового объекта в Python 3.X
hello world
>>> print 'hello world'       # Вывод строкового объекта в Python 2.X
hello world
```

Поскольку результаты выражений автоматически выводятся в строке интерактивной подсказки, часто даже не требуется использовать оператор `print` – необходимо просто набрать желаемые выражения и их результаты автоматически отобразятся:

```
>>> 'hello world'                                # Интерактивный вывод
'hello world'
```

Приведенный код точно не является поразительным результатом мастерства при разработке программного обеспечения, но он предназначен для иллюстрации поведения вывода. В действительности операция `print` представляет собой всего лишь удобное средство Python – оно обеспечивает простой интерфейс к объекту `sys.stdout` с незначительным стандартным форматированием. По сути, если вам нравится двигаться более трудным путем, чем необходимо, тогда можете записать операции вывода также и следующим образом (согласно главам 4 и 9; здесь опущено возвращаемое значение для Python 3.X):

```
>>> import sys                               # Вывод сложным способом
>>> sys.stdout.write('hello world\n')
hello world
```

В коде явно вызывается метод `write` объекта `sys.stdout` – атрибут, который предварительно устанавливается, когда Python начинает работать с открытым файловым объектом, подключенным к потоку вывода. Операция `print` скрывает большинство деталей подобного рода, предоставляя простой инструмент для решения несложных задач вывода.

Ручное перенаправление потока

Итак, почему я только что показал вам сложный способ вывода? Эквивалент вывода с помощью `sys.stdout` оказывается основой общепринятой методики в Python. В целом `print` и `sys.stdout` связаны непосредственно. Оператор:

```
print(X, Y)                                     # Или в Python 2.X: print X, Y
```

эквивалентен более длинному оператору:

```
import sys
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

в котором вручную выполняется преобразование в строку с помощью `str`, добавляется разделитель и новая строка посредством `+`, а также вызывается метод `write` потока вывода. Какому коду вы бы отдали предпочтение? (Говорю в надежде подчеркнуть дружественную к программистам природу операций `print`...)

Очевидно, что длинная форма сама по себе не настолько удобна для вывода. Тем не менее, полезно знать, что именно это и делают операции `print`, поскольку допускается *переназначить* `sys.stdout` на что-нибудь, отличающееся от стандартного потока вывода. Другими словами, такая эквивалентность обеспечивает возможность операциям `print` отправлять свой текст в другие места. Например:

```
import sys
sys.stdout = open('log.txt', 'a') # Перенаправление вывода в файл
...
print(x, y, x)                      # Отправляется в log.txt
```

Здесь мы переустанавливаем `sys.stdout` в открытый вручную файл по имени `log.txt`, который находится в рабочем каталоге и открыт в режиме добавления (так что мы можем дополнять его текущее содержимое). После переустановки каждая операция `print` везде в программе будет записывать свой текст в конец файла `log.txt`.

вместо исходного потока вывода. Операции `print` продолжают вызывать метод `write` объекта `sys.stdout` независимо от того, на что ссылается `sys.stdout`. Поскольку в процессе имеется лишь один модуль `sys`, подобное переназначение `sys.stdout` будет перенаправлять каждую операцию `print` повсюду в программе.

На самом деле, как объясняется во врезке “Что потребует внимания: `print` и `stdout`” в конце главы, вы можете переустанавливать `sys.stdout` даже в объект, вообще не имеющий отношения к файлам, при условии наличия в нем ожидаемого интерфейса — метода по имени `write` для получения аргумента со строкой выводимого текста. Когда такой объект является *классом*, выводимый текст можно произвольно направлять и обрабатывать согласно методу `write`, который вы написали самостоятельно.

Такой прием с переустановкой потока вывода может быть более полезен в программах, первоначально написанных с операторами `print`. Если вам заранее известно, что вывод должен попадать в файл, то вы можете взамен вызывать файловые методы записи. Однако при перенаправлении вывода программы, основанной на `print`, переустановка `sys.stdout` предоставляет удобную альтернативу изменению каждого оператора `print` или применению синтаксиса перенаправления, который базируется на командной оболочке системы.

В других ролях потоки могут быть переустановлены в объекты, которые отображают текст внутри всплывающих окон в графических пользовательских интерфейсах, раскрашивают его цветом в IDE-средах вроде IDLE и т.д. Это универсальная методика.

Автоматическое перенаправление потока

Хотя перенаправление выводимого текста путем переназначения `sys.stdout` является удобным инструментом, потенциальная проблема с кодом из предыдущего раздела связана с отсутствием прямой возможности восстановить исходный поток вывода, когда возникнет необходимость переключиться обратно после вывода в файл. Тем не менее, из-за того, что `sys.stdout` — всего лишь обычный файловый объект, вы всегда можете сохранить его и впоследствии восстановить⁶:

```
C:\code> c:\python33\python
>>> import sys
>>> temp = sys.stdout      # Сохранение с целью восстановления в будущем
>>> sys.stdout = open('log.txt', 'a')    # Перенаправление вывода в файл
>>> print('spam')           # Вывод направляется в файл, не сюда
>>> print(1, 2, 3)          # Сброс вывода на диск
>>> sys.stdout.close()     # Восстановление исходного потока
>>> sys.stdout = temp       # Теперь вывод снова виден
>>> print('back here')
back here
>>> print(open('log.txt').read())  # Результат предшествующих выводов
spam
1 2 3
```

Но, как вы могли заметить, подобного рода ручное сохранение и восстановление исходного потока вывода влечет за собой дополнительную работу.

⁶ В Python 2.X и 3.X вы также в состоянии использовать атрибут `__stdout__` из модуля `sys`, который ссылается на исходное значение `sys.stdout`, установленное во время начального запуска программы. Однако вам все равно придется восстанавливать `sys.stdout` в `sys.__stdout__`, чтобы возвратиться к исходному значению потока. Дополнительные сведения ищите в документации по модулю `sys`.

Поскольку такая потребность возникает довольно часто, стало доступным расширение `print`, которое делает ее ненужной.

В Python 3.X ключевое слово `file` позволяет одиночному вызову `print` отправлять свой текст методу `write` файлового (или подобного файлам) объекта без действительной переустановки `sys.stdout`. Так как перенаправление является временным, нормальные вызовы `print` продолжают выводить в исходный поток вывода.

В Python 2.X аналогичный эффект обеспечивает оператор `print`, начинающийся с символов `>>`, за которыми следует выходной файловый (или другой совместимый) объект. Скажем, приведенный ниже код снова отправляет выводимый текст в файл по имени `log.txt`:

```
log = open('log.txt', 'a')          # Python 3.X
print(x, y, z, file=log)           # Вывод в объект, подобный файлу
print(a, b, c)                    # Вывод в исходный stdout

log = open('log.txt', 'a')          # Python 2.X
print >> log, x, y, z             # Вывод в объект, подобный файлу
print a, b, c                      # Вывод в исходный stdout
```

Такие перенаправленные формы `print` удобны, когда в той же самой программе необходимо выводить *и* в файлы, *и* в стандартный поток вывода. Однако если вы используете эти формы, тогда удостоверьтесь, что предоставляемые им файловый объект (или объект, который имеет такой же метод `write`, как у файлового объекта), а не строку с именем файла. Взгляните на методику в действии:

```
C:\code> c:\python33\python
>>> log = open('log.txt', 'w')
>>> print(1, 2, 3, file=log)      # Для Python 2.X: print >> log, 1, 2, 3
>>> print(4, 5, 6, file=log)
>>> log.close()
>>> print(7, 8, 9)                # Для Python 2.X: print 7, 8, 9
7 8 9
>>> print(open('log.txt').read())
1 2 3
4 5 6
```

Показанные расширенные формы `print` также обычно применяются для вывода сообщений об ошибках в стандартный поток ошибок, доступный сценарию в виде предварительно открытого файлового объекта `sys.stderr`. Вы можете либо использовать его файловые методы `write` и вручную форматировать вывод, либо вызывать `print` посредством синтаксиса перенаправления:

```
>>> import sys
>>> sys.stderr.write('Bad!' * 8) + '\n'
Bad!Bad!Bad!Bad!Bad!Bad!
>>> print('Bad!' * 8, file=sys.stderr)
# В Python 2.X: print >> sys.stderr, 'Bad!' * 8
Bad!Bad!Bad!Bad!Bad!Bad!
```

Теперь, когда вы все знаете о перенаправлении вывода, эквивалентность операций `print` и файловых методов `write` должна стать вполне очевидной. В следующем взаимодействии производится вывод обоими способами в Python 3.X, а затем вывод перенаправляется во внешний файл и выведенный текст сравнивается:

```
>>> X = 1; Y = 2
>>> print(X, Y)                  # Вывод: легкий способ
1 2
```

```

>>> import sys          # Вывод: сложный способ
>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
1 2
4
>>> print(X, Y, file=open('temp1', 'w'))      # Перенаправление вывода в файл
>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\n')
# Отправка в файл вручную
4
>>> print(open('temp1', 'rb').read())    # Двоичный режим для вывода байтов
b'1 2\r\n'
>>> print(open('temp2', 'rb').read())
b'1 2\r\n'

```

Как видите, если только вы не получаете удовольствие от набора, операции `print` в большинстве случаев будут наилучшим вариантом для отображения текста. Еще один пример, который демонстрирует эквивалентность операций `print` и файловых методов `write`, предлагается в главе 18. Он эмулирует с помощью такого кодового шаблона функцию `print` из Python 3.X, чтобы предоставить ее универсальный эквивалент для использования в Python 2.X.

Вывод, нейтральный к версии

Наконец, на тот случай, когда необходимо, чтобы операции `print` работали в *обеих* линейках Python, имеется несколько вариантов. Это справедливо вне зависимости от того, пишете вы код Python 2.X, стремясь обеспечить его совместимость с Python 3.X, или же код Python 3.X, который направлен на поддержку также и Python 2.X.

Инструмент преобразования 2to3

Например, вы можете написать операторы `print` для Python 2.X и позволить сценарию преобразования 2to3 из Python 3.X автоматически перевести их в вызовы функции Python 3.X. Дополнительные сведения об указанном сценарии ищите в руководствах по Python 3.X; он пытается транслировать код Python 2.X для выполнения под управлением Python 3.X – полезный инструмент, но возможно делающий нечто большее, чем просто превращение операций `print` в нейтральные к версии. Связанный инструмент под названием 3to2 пытается выполнить обратную трансляцию: он преобразует код Python 3.X для выполнения под управлением Python 2.X; дополнительная информация приведена в приложении В второго тома.

Использование `from __future__ import print_function`

В качестве альтернативы можно записывать вызовы функции `print` из Python 3.X в коде, предназначенном для выполнения в Python 2.X, для чего включить вариант с вызовом функции посредством следующего оператора, который должен находиться в начале сценария или где угодно в интерактивном сеансе:

```
from __future__ import print_function
```

Показанный оператор изменяет Python 2.X для полной поддержки функции `print` из Python 3.X. В итоге появляется возможность применения функции `print` из Python 3.X и вдобавок не придется изменять операции `print`, если позже будет решено перенести код в Python 3.X.

Ниже приведены два замечания относительно использования.

- Этот оператор попросту игнорируется, когда встречается в коде, выполняемом под управлением Python 3.X – он вовсе не мешает, если включен в код Python 3.X для совместимости с Python 2.X.
- Этот оператор должен находиться в начале каждого файла, где применяются операции print в Python 2.X – поскольку он действует только на один файл, импортирования другого файла, включающего данный оператор, будет недостаточно.

Нейтрализация отличий в отображении с помощью кода

Также следует отметить, что простые операции print, подобные показанным в первой строке табл. 11.5, работают в *любой* версии Python – поскольку любое выражение может быть заключено в круглые скобки, мы всегда можем в Python 2.X сделать вид, что вызываем функцию print из Python 3.X, добавив внешние круглые скобки. Главный недостаток такого подхода связан с тем, что он создает *кортеж*, если выводимых объектов более одного или вообще нет – объекты будут выводиться в добавочных круглых скобках. Скажем, в Python 3.X в круглых скобках вызова можно перечислять любое количество объектов:

```
C:\code> c:\python33\python
>>> print('spam')                                # Синтаксис вызова функции print в Python 3.X
spam
>>> print('spam', 'ham', 'eggs')    # Множество аргументов
spam ham eggs
```

Первый оператор работает точно так же и в Python 2.X, но второй создает кортеж в выводе:

```
C:\code> c:\python27\python
>>> print('spam')                                # Оператор print в Python 2.X,
                                                #   заключающие круглые скобки
spam
>>> print('spam', 'ham', 'eggs')    # На самом деле это объект кортежа!
('spam', 'ham', 'eggs')
```

То же самое происходит, когда не указывается *ни одного* объекта, чтобы инициировать перевод строки: Python 2.X отобразит кортеж, если только не вывести пустую строку:

```
c:\code> py -2
>> print()                                     # Просто перевод строки в Python 3.X
()
>>> print('')                                    # Перевод строки в Python 2.X и 3.X
```

Строго говоря, в ряде случаев выводы могут отличаться более чем только добавочными заключающими круглыми скобками в Python 2.X. Если вы внимательнее присмотритесь к предыдущим результатам, то также заметите, что строки выводятся в *заключающих кавычках* только в Python 2.X. Причина в том, что объекты могут выводиться по-разному, когда *вложены* в другой объект, нежели когда они являются элементами верхнего уровня. Формально вложенные экземпляры отображаются с помощью repr, а объекты верхнего уровня посредством str – два альтернативных формата, которые отмечались в главе 5.

Здесь это означает просто добавочные кавычки вокруг строк, вложенных в кортеж, который в Python 2.X создается для вывода множества элементов, помещенных

в круглые скобки. Тем не менее, отображение вложенных объектов может отличаться еще больше для объектов других типов, особенно для объектов классов, которые определяют альтернативное отображение с помощью *перегрузки операций* – данную тему мы рассмотрим в целом в части VI и в частности в главе 30.

Чтобы обеспечить подлинную переносимость, не включая повсюду операции `print` из Python 3.X, и обойти отличия в отображении вложенных экземпляров, вы всегда можете форматировать выводимую строку как одиничный объект для унификации ее отображения разными версиями, используя выражение или вызов метода форматирования строк либо другие инструменты, которые были описаны в главе 7:

```
>>> print('%s %s %s' % ('spam', 'ham', 'eggs'))
spam ham eggs
>>> print('{0} {1} {2}'.format('spam', 'ham', 'eggs'))
spam ham eggs
>>> print('answer: ' + str(42))
answer: 42
```

Разумеется, если у вас есть возможность работать исключительно в Python 3.X, то вы можете совершенно не думать о таких сопоставлениях, но многим программистам на Python в течение некоторого времени придется минимум сталкиваться, а то и писать код Python 2.X. Мы будем применять `from __future__ import` и нейтральный к версии код для обеспечения переносимости Python 2.X/3.X во многих примерах, приводимых в настоящей книге.



Я использую вызовы функции `print` из Python 3.X повсеместно в книге. Я часто делаю операции `print` нейтральными к версии и обычно предупреждаю, когда результаты могут отличаться в Python 2.X, но иногда не предупреждаю, поэтому считайте данную врезку всеохватывающим предупреждением. Если в Python 2.X вы видите в выводимом тексте добавочные круглые скобки, тогда либо уберите круглые скобки из операторов `print`, импортируйте операции `print` из Python 3.X посредством `from __future__ import`, перепишите операции `print` с применением обрисованной здесь схемы нейтральности к версии либо научитесь любить излишний текст.

Что потребует внимания: `print` и `stdout`

Эквивалентность операции `print` и метода `write` объекта `sys.stdout` важна. Она делает возможным переназначение `sys.stdout` в любой определяемый пользователем объект, который предоставляет тот же самый метод `write`, что и файловые объекты. Поскольку оператор `print` всего лишь отправляет текст методу `sys.stdout.write`, вы можете захватывать выводимый текст в своих программах, присваивая `sys.stdout` объект, метод `write` которого обрабатывает текст произвольными способами.

Скажем, выводимый текст можно передать окну графического пользовательского интерфейса либо направить его по многим назначениям, определяя объект с методом `write`, который не требует маршрутизации. Пример такого трюка будет приведен во время исследования классов в части VI книги, но абстрактно он выглядит следующим образом:

```
class FileFaker:
    def write(self, string):
        # Делать что-то с выводимым текстом в строке
```

```
import sys
sys.stdout = FileFaker()
print(someObjects) # Отправка методу write класса
```

Прием работает, потому что `print` – это то, что в следующей части книги мы будем называть *полиморфной* операцией; ее не заботит, чем является `sys.stdout`, а лишь то, имеется ли метод (т.е. интерфейс) по имени `write`. Такое перенаправление на объекты становится даже более простым в случае использования ключевого аргумента `file` в Python 3.X и расширенной посредством `>>` формы `print` в Python 2.X, потому что явно переустанавливать `sys.stdout` не требуется – нормальные операции `print` будут по-прежнему направляться потоку `stdout`:

```
myobj = FileFaker() # Python 3.X: перенаправление
# на объект для одной операции print
print(someObjects, file=myobj) # Не переустанавливает sys.stdout
myobj = FileFaker() # Python 2.X: тот же самый эффект
print >> myobj, someObjects # Не переустанавливает sys.stdout
```

Встроенная функция `input` в Python 3.X (называемая `raw_input` в Python 2.X) читает из файла `sys.stdin`, так что перехватывать запросы на чтение можно похожим образом, применяя классы, которые взамен реализуют методы, подобные файловому методу `read`. За дополнительными сведениями об этой функции обращайтесь к примерам использования `input` и `while` в главе 10.

Обратите внимание, что поскольку выводимый текст направляется в поток `stdout`, это также является способом вывода HTML-страниц ответов в CGI-сценариях, применяемых в веб-сети, и позволяет обычным образом перенаправлять ввод и вывод сценариев Python в командной оболочке операционной системы:

```
python script.py < inputfile > outputfile
python script.py | filterProgram
```

Инструменты перенаправления операций `print` в Python по существу представляют собой альтернативы таких синтаксических форм командной оболочки на чистом Python. Информацию о CGI-сценариях и синтаксических формах командной оболочки ищите на других ресурсах.

Резюме

В главе мы начали углубленное рассмотрение операторов Python с исследования присваиваний, выражений и операций вывода. Хотя эти операторы в целом просты в использовании, они имеют альтернативные формы, которые при своей необязательности часто удобны на практике – скажем, операторы дополненного присваивания и перенаправляющие формы операций `print` позволяют избежать написания определенного кода вручную. Попутно также проводились исследования синтаксиса именования переменных, методик перенаправления потоков и распространенных ошибок, которых следует избегать, таких как присваивание результата вызова метода `append` обратно переменной.

В следующей главе мы продолжим наш тур по операторам заполнением недостающих деталей, касающихся оператора `if`, который является главным инструментом выбора в Python; там мы также более глубоко рассмотрим синтаксическую модель Python и подробнее обсудим поведение булевых выражений. Однако прежде чем двигаться дальше, проверьте свои знания, ответив на контрольные вопросы главы.

Проверьте свои знания: контрольные вопросы

1. Назовите три способа, с помощью которых вы можете присвоить трем переменным одно и то же значение.
2. Почему необходимо соблюдать осторожность в случае присваивания трем переменным изменяемого объекта?
3. Что не так с оператором `L = L.sort()`?
4. Как вы могли бы применить операцию `print` для отправки текста во внешний файл?

Проверьте свои знания: ответы

1. Вы можете использовать групповое присваивание (`A = B = C = 0`), присваивание последовательности (`A, B, C = 0, 0, 0`) или множество операторов присваивания в трех отдельных строках (`A = 0, B = 0` и `C = 0`). При последнем способе, как объяснялось в главе 10, вы также можете поместить операторы в одну строку, разделяя их точками с запятой (`A = 0; B = 0; C = 0`).

2. Если вы выполните присваивание следующим образом:

```
A = B = C = []
```

то все три имени будет ссылаться на тот же самый объект, так что изменение на месте через одно имя (например, `A.append(99)`) повлияет на остальные. Это справедливо только для модификации на месте изменяемых объектов вроде списков и словарей; к неизменяемым объектам, таким как числа и строки, проблема не относится.

3. Списковый метод `sort` подобен `append` в том, что он вносит изменение на месте в список, на котором вызывается — метод `sort` возвращает `None`, а не измененный список. Присваивание обратно `L` устанавливает `L` в `None`, а не в отсортированный список. Как обсуждалось ранее (скажем, в главе 8) и будет обсуждаться позже в книге, более новая встроенная функция `sorted` сортирует любую последовательность и возвращает новый список с результатом сортировки; поскольку это не изменение на месте, его результат можно осмысленно присвоить имени.
4. Чтобы обеспечить вывод в файл для одиночной операции `print`, вы можете использовать форму вызова `print(X, file=F)` в Python 3.X, применить расширенную форму оператора `print >> file`, `X` в Python 2.X или присвоить `sys.stdout` вручную открытый файл перед операцией `print` и впоследствии восстановить исходное состояние. Вы также можете перенаправить весь выводимый текст программы в файл с помощью специального синтаксиса в командной строке системы, но такой прием выходит за рамки Python.

Проверки `if` и правила синтаксиса

В этой главе рассматривается оператор `if` языка Python, который является главным оператором, используемым для выбора варианта действия на основе результатов проверки. Поскольку мы впервые обратимся к *составным операторам*, т.е. операторам, в которые встроены другие операторы, то также более детально исследуем общие концепции введенной в главе 10 синтаксической модели операторов Python. Из-за того, что оператор `if` привносит понятие проверок, мы также обсудим булевские выражения, раскроем “тернарное” выражение `if` и взглянем на ряд подробностей, касающихся проверок истинности в целом.

Операторы `if`

Формулируя упрощенно, оператор `if` языка Python выбирает действия для выполнения. Вместе со своим эквивалентом в виде выражения он является основным инструментом выбора в Python и представляет большой объем логики, которой обладает программа Python. Вдобавок это наш первый составной оператор. Подобно всем составным операторам Python оператор `if` способен содержать другие операторы, включая дополнительные `if`. На самом деле Python позволяет компоновать операторы в программе последовательно (так что они выполняются друг за другом) и с произвольной глубиной вложения (чтобы они выполнялись только в случае удовлетворения определенных условий, таких как применяемые при выборе и в цикле).

Общий формат

Оператор `if` языка Python является типичным оператором `if` из большинства процедурных языков. Он принимает форму проверки `if`, за которой следует одна или большее количество проверок `elif` (“else if”) и финальный необязательный блок `else`. С каждой проверкой и частью `else` связан блок вложенных операторов, расположенный с отступом относительно строки заголовка. Когда оператор `if` запускается, Python выполняет блок кода, ассоциированный с первой проверкой, которая дает истинный результат, или блок `else`, если все проверки имели ложные результаты.

Ниже показана общая форма оператора if:

```
if проверка1:      # Проверка if
    оператор1      # Связанный блок
elif проверка2:    # Необязательные elif
    оператор2
else:              # Необязательный else
    оператор3
```

Элементарные примеры

В целях демонстрации давайте посмотрим на несколько примеров оператора if в работе. Все его части необязательны кроме начальной проверки if и ассоциированных с ней операторов. Таким образом, в простейшем случае остальные части опускаются:

```
>>> if 1:
...     print('true')
...
true
```

Обратите внимание, что приглашение к вводу изменяется на . . . для набора строк продолжения в базовой интерактивной подсказке; в IDLE взамен происходит переход на следующую строку с отступом (для возврата нужно нажать клавишу забоя). Пустая строка (получаемая двукратным нажатием клавиши <Enter>) заканчивает и запускает весь оператор. Вспомните, что 1 — это булевское значение “истина” (как вы увидите позже, его эквивалентом является слово True), так что проверка в операторе всегда проходит. Для обработки ложного результата понадобится добавить часть else:

```
>>> if not 1:
...     print('true')
... else:
...     print('false')
...
false
```

Множественное ветвление

А вот пример более сложного оператора if, содержащего все необязательные части:

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print("shave and a haircut")
... elif x == 'bugs':
...     print("what's up doc?")
... else:
...     print('Run away! Run away!')
...
Run away! Run away!
```

Приведенный многострочный оператор простирается от строки if до блока, вложенного внутрь части else. Когда он запускается, Python выполняет операторы, вложенные внутрь первой проверки, которая дает истинный результат, или часть else, если все проверки показали ложные результаты (как в данном примере). На практике части elif и else могут быть опущены, а в каждой части допускается указывать более одного вложенного оператора. Обратите внимание, что слова if, elif и else связываются друг с другом по факту выравнивания по вертикали с одинаковыми отступами.

Если вы использовали языки наподобие C или Pascal, тогда вам небезынтересно будет узнать, что в Python отсутствует оператор вроде `switch` или `case`, который выбирал бы действие на основе значения переменной. Взамен множественное ветвление обычно записывается в виде серии проверок `if/elif`, как в предыдущем примере, и иногда реализуется путем индексирования словарей или поиска в списках. Поскольку словари и списки можно динамически создавать на стадии выполнения, временами они оказываются более гибкими, чем логика `if`, жестко закодированная в сценарии:

```
>>> choice = 'ham'  
>>> print({'spam': 1.25,           # Аналог switch на основе словаря  
...      'ham': 1.99,            # Для получения стандартного значения  
...      'eggs': 0.99,          # используется has_key или get  
...      'bacon': 1.10}[choice])  
1.99
```

Хотя осознание этого поначалу может потребовать некоторого времени, показанный словарь реализует множественное ветвление – индексация по ключу `choice` обеспечивает переход к одному из набора значений во многом подобно оператору `switch` в C. Почти эквивалентный, но более многословный оператор `if` языка Python мог бы выглядеть следующим образом:

```
>>> if choice == 'spam':        # Эквивалентный оператор if  
...     print(1.25)  
... elif choice == 'ham':  
...     print(1.99)  
... elif choice == 'eggs':  
...     print(0.99)  
... elif choice == 'bacon':  
...     print(1.10)  
... else:  
...     print('Bad choice')  
...  
1.99
```

Несмотря на возможно лучшую читабельность, его потенциальный недостаток в том, что если не считать оформления такого оператора `if` в виде строки и выполнения с помощью `eval` или `exec`, то создать его во время выполнения не настолько легко, как словарь. В более динамических программах структуры данных зачастую добавляют гибкости.

Поддержка стандартных значений

В приведенном выше операторе `if` обратите внимание на конструкцию `else`, которая предназначена для обработки стандартного случая, когда совпадения по ключу отсутствуют. Как объяснялось в главе 8, стандартные значения словарей можно записывать с помощью выражений `in`, вызовов метода `get` или перехвата исключений посредством оператора `try`, представленного в предыдущей главе. Те же самые методики можно применять здесь для кодирования стандартного действия при множественном ветвлении, основанном на словаре. В качестве обзора в контексте этого сценария использования далее показано, как схема `get` работает со стандартными значениями:

```
>>> branch = {'spam': 1.25,  
...             'ham': 1.99,  
...             'eggs': 0.99}
```

```
>>> print(branch.get('spam', 'Bad choice'))
1.25
>>> print(branch.get('bacon', 'Bad choice'))
Bad choice
```

Того же результата можно добиться с применением проверки членства `in` в операторе `if`:

```
>>> choice = 'bacon'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Bad choice')
...
Bad choice
```

Наконец, оператор `try` предлагает обобщенный способ поддержки стандартных значений путем перехвата и обработки исключений, которые возникли бы в противном случае (дополнительные сведения об исключениях можно почерпнуть из обзора в главе 11 и полного описания в части VII):

```
>>> try:
...     print(branch[choice])
... except KeyError:
...     print('Bad choice')
...
Bad choice
```

Поддержка более сложных действий

Словари хороши для связывания значений с ключами, но как насчет более замысловатых действий, которые можно записывать в блоках операторов, ассоциированных с операторами `if`? В части IV вы узнаете, что словари могут также содержать функции для представления более сложных действий ветвления и реализации обобщенных таблиц переходов. Такие функции выглядят как значения словаря, они могут быть записаны как имена функций или внутристорчные выражения `lambda` и могут вызываться добавлением круглых скобок для инициализации их действий. Ниже приводится абстрактный шаблон, но дождитесь возвращения к данной теме в главе 19 после ознакомления с определением функций:

```
def function(): ...
def default(): ...

branch = {'spam': lambda: ...,
          'ham': function,
          'eggs': lambda: ...}

branch.get(choice, default)()
```

Хотя множественное ветвление на основе словаря удобно в программах, которые имеют дело с динамическими данными, вероятно большинство программистов обнаружат, что написание оператора `if` является самым прямолинейным способом выполнения множественного ветвления. В качестве эмпирического правила запомните, что в случае сомнений оставайтесь на стороне простоты и читабельности; такой путь соответствует стилю программирования в духе Python.

Снова о синтаксисе Python

Синтаксическая модель Python была представлена в главе 10. Теперь, когда мы пошли к более крупным операторам вроде `if`, в настоящем разделе предлагается обзор и развитие ранее описанных идей синтаксиса. В целом Python обладает простым синтаксисом на основе операторов. Однако существует несколько характеристик, о которых следует знать.

- Операторы выполняются друг за другом, если только не указано иное. Python обычно выполняет операторы из файла или вложенного блока в порядке с первого до последнего как последовательность, но операторы, подобные `if` (а также циклы и исключения), заставляют интерпретатор совершать переходы в коде. Из-за того, что путь Python через программу называется потоком управления, операторы вроде `if`, которые воздействуют на него, часто называют операторами потока управления.
- Границы блоков и операторов определяются автоматически. Как вы уже видели, в Python отсутствуют какие-либо скобки или ограничители “начала/конца” в блоках кода; для группирования операторов во вложенный блок используются их отступы относительно заголовка. Подобным же образом операторы Python, как правило, не завершаются точкой с запятой; замен конец строки обычно обозначает окончание оператора, записанного в этой строке. В качестве особого случая с помощью специального синтаксиса операторы могут занимать несколько строк и размещаться в одной строке.
- Составные операторы = заголовок + : + операторы с отступом. Все составные операторы Python – с вложенными операторами – следуют одному шаблону: строка заголовка, завершающаяся двоеточием, за которой находится один или больше операторов, обычно записанных с отступом относительно заголовка. Операторы с отступом называются блоком (либо иногда набором). В операторе `if` конструкции `elif` и `else` являются частью `if`, но также и строками заголовков с собственными вложенными блоками. Как особый случай блоки могут находиться в той же строке, что и заголовок, если представляют собой невложенный код.
- Пустые строки, пробелы и комментарии обычно игнорируются. Пустые строки являются необязательными и игнорируются в файлах, но не в интерактивной подсказке, где они завершают составные операторы. Пробелы внутри операторов и выражений почти всегда игнорируются (за исключением ситуации, когда они находятся внутри строковых литералов и когда применяются для отступа). Комментарии всегда игнорируются: они начинаются с символа # (не внутри строкового литерала) и простираются до конца текущей строки.
- Строки документации игнорируются, но сохраняются и отображаются инструментами. Python поддерживает дополнительную форму комментариев, называемую строками документации, которые в отличие от комментариев # запоминаются во время выполнения для инспектирования. Строки документации – это просто строки, отображаемые в верхней части файлов программ и ряда операторов. Python игнорирует их содержимое, но они автоматически присоединяются к объектам во время выполнения и могут отображаться посредством инструментов документации, подобных PyDoc. Строки документации входят в состав более широкой стратегии документирования Python и раскрываются в последней главе данной части книги.

Как вы уже видели, объявление типов переменных в Python не производится; этот факт сам по себе делает синтаксис языка намного проще, нежели то, к чему вы могли привыкнуть. Тем не менее, для большинства новых пользователей отсутствие скобок и точек с запятой, используемых для отметки блоков и операторов во множестве других языков, выглядит самой нестандартной синтаксической особенностью Python, поэтому давайте более точно выясним, что она означает.

Ограничители блоков: правила отступов

В главе 10 упоминалось, что Python определяет границы блоков автоматически, по отступам строк — т.е. пустому пространству слева от кода. Все операторы с отступами на одинаковое расстояние вправо принадлежат тому же самому блоку кода. Другими словами, операторы внутри блока выровнены по вертикали как в столбце. Блок заканчивается, когда встречается конец файла или строка с меньшим отступом, и более глубоко вложенный блок просто смещается дальше вправо, чем операторы во включающем его блоке. В ряде случаев, которые мы исследуем позже, в строке заголовка могут находиться тела составных операторов, но в большинстве ситуаций они располагаются с отступом под ней.

Например, на рис. 12.1 изображена структура блоков следующего кода:

```
x = 1
if x:
    y = 2
    if y:
        print('block2')
    print('block1')
print('block0')
```

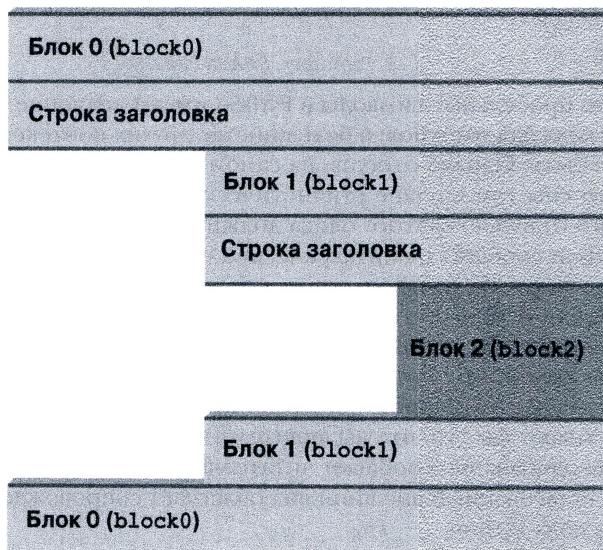


Рис. 12.1. Вложенные блоки кода: вложенный блок начинается с оператора, смещенного дальше вправо, и заканчивается либо на операторе с меньшим отступом, либо в случае конца файла

Код содержит три блока: первый (код верхнего уровня файла) вообще не имеет отступа, второй (внутри внешнего оператора `if`) смещен на четыре пробела и третий (оператор `print` под вложенным `if`) смещен на восемь пробелов.

В общем случае код верхнего уровня (невложенный) обязан начинаться в столбце 1. Вложенные блоки допускается начинать в любом столбце; отступ может состоять из любого количества пробелов и табуляций при условии, что он одинаков для всех операторов в отдельно взятом блоке. То есть Python совершенно не заботит, *каким образом* вы смещаете свой код; для него важно лишь то, чтобы отступы делались согласованно. Четыре пробела или одна табуляция на уровень отступа является распространенным соглашением, но абсолютных стандартов в мире Python не существует.

На практике снабжение кода отступами вполне естественно. Скажем, в приведенном далее (возможно, нелепом) фрагменте кода демонстрируются часто допускаемые ошибки с отступами в Python:

```
x = 'SPAM'                                # Ошибка: первая строка имеет отступ
if 'rubbery' in 'shrubbery':
    print(x * 8)
        x += 'NI'                            # Ошибка: неожиданный отступ
    if x.endswith('NI'):
        x *= 2
    print(x)                                # Ошибка: несогласованный отступ
```

Ниже показана версия этого кода с надлежащими отступами – даже для такого искусственного примера правильные отступы делают намерение кода более явным:

```
x = 'SPAM'
if 'rubbery' in 'shrubbery':
    print(x * 8)                          # Выводит 8 раз SPAM
    x += 'NI'
if x.endswith('NI'):
    x *= 2
print(x)                                  # Выводит SPAMNISPAMNI
```

Важно знать, что пробельные символы в Python имеют значение только когда применяются слева от кода для отступов; в большинстве других контекстов пробел может быть, а может и не быть. Однако отступы на самом деле являются частью синтаксиса Python, а не просто советом в плане стилистического оформления кода: все операторы внутри любого отдельно взятого блока должны быть смещены на тот же самый уровень или же Python сообщит о синтаксической ошибке. Так сделано преднамеренно – поскольку явно отмечать начало и конец вложенного блока не требуется, синтаксическое нагромождение, присутствующее в ряде других языков, в Python излишне.

Как было показано в главе 10, включение отступов в синтаксическую модель также обеспечивает согласованность, которая представляет собой важнейший компонент читабельности в языках структурного программирования вроде Python. Иногда синтаксис Python описывают как “то, что вы видите, будет тем, что вы получите” – отступ каждой строки кода однозначно сообщает читателям, с чем она ассоциирована. Такой единообразный и согласованный внешний вид облегчает сопровождение и многократное использование кода Python.

На практике применять отступы проще, чем первоначально может вытекать из связанных с ними деталей, и они заставляют код отражать свою логическую структуру. Код с согласованными отступами всегда удовлетворяет правилам Python. Кроме того, большинство текстовых редакторов (в том числе IDLE) упрощают следование модели отступов Python, автоматически смещающая код по мере его набора.

Избегайте смещивания табуляций и пробелов: новая проверка ошибок в Python 3.X

Запомните эмпирическое правило: хотя вы можете использовать для отступов пробелы или табуляции, их *смещивание* внутри блока обычно не будет удачной идеей — применяйте либо то, либо другое. Формально табуляция считается достаточным количеством пробелов, чтобы сместить текущую строку на расстояние, кратное 8, и код будет работать в случае согласованного смещивания табуляций и пробелов. Тем не менее, такой код может быть сложнее изменять. Хуже того, смещивание табуляций и пробелов затрудняет чтение кода целиком, не говоря уже о правилах синтаксиса Python — табуляции в редакторе сменившего вас программиста могут выглядеть совсем не так, как в вашем редакторе.

Именно по указанным выше причинам Python 3.X выдает ошибку, когда в сценарии внутри какого-то блока несогласованно смещиваются табуляции и пробелы для отступов (т.е. способом, который делает отступы зависящими от эквивалента табуляции в пробелах). Python 2.X разрешает запускать такие сценарии, но имеет флаг командной строки `-t`, позволяющий предупреждать о несогласованном использовании табуляций, и флаг `-tt`, который обеспечит генерацию ошибок в таком коде (эти флаги можно применять внутри командной строки, подобной `python -t main.py`, в окне командной оболочки). Новая проверка ошибок Python 3.X эквивалентна запуску Python 2.X с флагом `-tt`.

Ограничители операторов: строки и продолжения

Оператор в Python обычно завершается в конце строки, где он находится. Однако когда оператор слишком длинный, чтобы уместиться в одной строке, то есть несколько специальных правил, с помощью которых его можно распространить на множество строк.

- **Операторы могут занимать несколько строк, если они продолжаются после открытой скобки из синтаксической пары.** Python позволяет продолжить набор оператора в следующей строке, если вы заключите его в пару скобок `()`, `{}` или `[]`. Например, выражения в круглых скобках, а также литералы словарей и списков могут распространяться на любое количество строк; оператор не закончится до тех пор, пока интерпретатор Python не достигнет строки, в которой вы наберете закрывающую скобку пары `(`, `}` или `]`). Строки продолжения — строки 2 и далее в операторе — могут начинаться с любым желаемым отступом, но вы должны стараться по возможности вертикально выравнивать их ради читабельности. Правило с открытой скобкой из пары охватывает также включения множеств и словарей в Python 3.X и 2.7.
- **Операторы могут занимать несколько строк, если они завершаются обратной косой чертой.** Хотя это несколько устаревшая возможность, которую использовать не рекомендуется, но если оператор необходимо распространить на несколько строк, то можно добавить в конец предыдущей строки обратную косую черту (символ `\`, не встроенный в строковый литерал или комментарий), чтобы указать на продолжение оператора в следующей строке. Из-за того, что продолжать можно также путем помещения в круглые скобки большинства конструкций, обратные косые черты в наши дни применяются редко. Вдобавок такой подход подвержен ошибкам: если случайно забыть о наборе символа `\`, то обычно возникает синтаксическая ошибка, а следующая строка даже может молча (т.е. без выдачи предупреждения) неправильно воспринята как новый оператор, приводя к непредсказуемым результатам.

- **Специальные правила для строковых литералов.** Как было показано в главе 7, блочные строки в укрупненных кавычках предназначены для нормального охвата множества строк. Также в главе 7 говорилось о том, что примыкающие друг к другу строковые литералы неявно соединяются; когда это используется в сочетании с упомянутым ранее правилом с открытой скобкой из пары, помещение такой конструкции в круглые скобки позволяет распространить ее на множество строк.
- **Другие правила.** Есть несколько других моментов, касающихся ограничителей операторов, о которых полезно знать. Хотя и необычно, но оператор можно завершать посредством точки с запятой — иногда такое соглашение применяется для уплотнения множества простых (несоставных) операторов в одну строку. Кроме того, в файле где угодно могут присутствовать комментарии и пустые строки; комментарии (начинающиеся с символа #) завершаются в конце строки, в которой находятся.

Несколько особых случаев

Ниже показано, как выглядит строка продолжения, в которой используется только что описанное правило с открытой скобкой из синтаксической пары. Конструкции с ограничителями, такие как списки в квадратных скобках, могут распространяться на любое количество строк:

```
L = ["Good",
      "Bad",
      "Ugly"]      # Открытая скобка из пары позволяет охватывать много строк
```

Прием также работает для чего угодно в круглых скобках (выражений, аргументов функций, заголовков функций, кортежей и генераторных выражений) и в фигурных скобках (словарей, а также литералов множеств и включений множеств и словарей в Python 3.X/2.7). Часть таких инструментов мы обсудим в последующих главах, но правило с открытой скобкой из синтаксической пары естественным образом охватывает большинство конструкций, которые на практике располагаются в нескольких строках.

Если вам нравится продолжение с применением обратной косой черты, тогда можете поступать так, но это не является общепринятой практикой в Python:

```
if a == b and c == d and \
   d == e and f == g:
    print('olde') # Обратная косая черта делает возможным продолжение...
```

Поскольку любое выражение может быть заключено в круглые скобки, взамен обычно можно использовать методику с открытой скобкой из синтаксической пары, если необходимо разнести код на множество строк — нужно просто поместить часть оператора в круглые скобки:

```
if (a == b and c == d and
    d == e and e == f):
    print('new') # Но круглые скобки обычно делают то же самое, и они нагляднее
```

На самом деле большинство разработчиков на Python в целом не одобряют обратные косые черты, потому что их слишком легко не заметить или вообще опустить. В приведенном ниже коде x присваивается значение 10 при наличии обратной косой черты, как и задумывалось; тем не менее, если обратную косую черту случайно опустить, то x получит значение 6, и ни о каких ошибках сообщаться не будет (+4 само по себе является допустимым выражением).

В реальной программе с более сложным присваиванием отсутствие обратной косой черты могло бы стать источником крайне опасных дефектов¹:

```
x = 1 + 2 + 3 \          # Отсутствие \ делает результат совершенно другим!
+4
```

Еще один особый случай касается того, что Python позволяет записывать несколько несоставных операторов (т.е. операторов без вложенных операторов) в одной строке, разделяя их точками с запятой. Некоторые кодировщики используют такую форму для экономии пространства в файле программы, но в большинстве ситуаций код будет более читабельным, если размещать в строке по одному оператору:

```
x = 1; y = 2; print(x)    # Более одного простого оператора
```

В главе 7 вы узнали, что строковые литералы в устроенных кавычках также размещаются в нескольких строках. Вдобавок, если два строковых литерала находятся рядом друг с другом, то они соединяются, как будто между ними находится операция `+`; в случае применения в сочетании с правилом с открытой скобкой из синтаксической пары заключение в круглые скобки позволяет такой форме распространяться на множество строк. Скажем, первый оператор из показанных далее вставляет символы новой строки в местах разрыва строк и присваивает переменной `S` строку `'\naaaa\nbbbb\ncccc'`, а второй выполняет явную конкатенацию и присваивает `S` строку `'aaaabbbbcccc'`. Как мы видели в главе 7, во второй форме комментарии `#` игнорируются, но в первой форме они включаются в строку:

```
S = """
aaaa
bbbb
cccc"""

S = ('aaaa'
      'bbbb'           # Комментарий здесь игнорируется
      'cccc')
```

Наконец, Python дает возможность переносить тело составного оператора в строку заголовка при условии, что оно содержит только простые (несоставные) операторы. Чаще всего вы будете это видеть для простых операторов `if` с одиночным действием проверки, как в интерактивном цикле, который был реализован в главе 10:

```
if 1: print('hello')      # Простой оператор в строке заголовка
```

Вы можете комбинировать описанные особые случаи и в итоге получать код, который будет трудным для восприятия, но я не рекомендую делать это; в качестве эмпирического правила старайтесь размещать каждый оператор в отдельной строке и делайте отступы для всех блоков кроме простейших. Спустя полгода вы будете счастливы, что поступали так.

¹ Откровенно говоря, слегка удивительно, что продолжение посредством обратной косой черты не было изъято из версии Python 3.0, учитывая большой масштаб других изменений! Список того, что было удалено в версии Python 3.0, приведен в таблице изменений Python 3.0 в приложении В второго тома; некоторые из них выглядят довольно безобидными в сравнении с опасностями, присущими продолжению с помощью обратной косой черты. Но в то же время цель данной книги — обучение Python, а не популистское возмущение, поэтому мой добрый совет прост: не поступайте так. Как правило, в новом коде Python вы должны избегать использования продолжений посредством обратной косой черты, даже если приобрели такую привычку во времена программирования на языке C.

Значения истинности и булевские проверки

Понятия сравнения, равенства и значений истинности были представлены в главе 9. Поскольку `if` – первый рассмотренный оператор, который действительно использует результаты проверки, здесь мы расширим некоторые из лежащих в его основе идей. В частности, булевские операции Python слегка отличаются от своих аналогов в языках, подобных С. В Python:

- все объекты имеют неотъемлемое булевское значение истины или лжи;
- любое ненулевое число или непустой объект является истинным;
- нулевые числа, пустые объекты и специальный объект `None` считаются ложными;
- сравнения и проверки на равенство применяются к структурам данных рекурсивно;
- сравнения и проверки на равенство возвращают `True` или `False` (специальные версии 1 и 0);
- булевские операции `and` и `or` возвращают объект истинного или ложного операнда;
- булевские операции прекращают оценку (“укорачиваются”), как только результат становится известным.

Оператор `if` выполняет действие на значениях истинности, но булевские операции используются для объединения результатов других проверок более развитыми способами, чтобы получить новые значения истинности. Более формально в Python существуют три булевых операции выражений.

<code>X and Y</code>	Дает истину, если истинными являются и X, и Y
<code>X or Y</code>	Дает истину, если истинным является либо X, либо Y
<code>not X</code>	Дает истину, если X ложно (выражение возвращает <code>True</code> или <code>False</code>)

Здесь X и Y могут быть любыми значениями истинности или любым выражением, возвращающим значение истинности (например, проверкой равенства, сравнением диапазона и т.д.). Булевские операции в Python набираются как слова (в отличие от `&&`, `||` и `!` в языке С). Кроме того, булевские операции `and` и `or` в Python возвращают истинный или ложный объект, а не значение `True` или `False`. Давайте рассмотрим несколько примеров, чтобы увидеть, как они работают:

```
>>> 2 < 3, 3 < 2    # Меньше чем: возвращает True или False (1 или 0)
(True, False)
```

Сравнения относительных величин вроде показанных возвращают значения `True` или `False` в качестве своих результатов истинности, которые, как объяснялось в главах 5 и 9, на самом деле представляют собой всего лишь специальные версии целых чисел 1 и 0 (они выводят себя иначе, но во всем остальном ничем не отличаются).

С другой стороны, операции `and` и `or` всегда возвращают объект – либо объект в левой части операции, либо объект в правой ее части. Проверка их результатов в `if` или других операторах проходит ожидаемым образом (вспомните, что каждый объект по своему существу является истинным или ложным), но мы не получаем обратно простое значение `True` или `False`.

При проверке `or` объекты операндов оцениваются слева направо, и возвращается первый истинный объект. Кроме того, Python останавливается на первом найденном истинном объекте. Обычно это называют укороченной оценкой, т.к. определение результата остатка выражения прекращается, как только результат становится известным:

```
>>> 2 or 3, 3 or 2      # Возвращает левый операнд, если он истинный
(2, 3)                  # В противном случае возвращает правый операнд
                        # (истинный или ложный)
>>> [] or 3
3
>>> [] or {}
{}
```

В первой строке предыдущего примера оба операнда (2 и 3) истинны (т.к. они не нулевые), поэтому Python останавливается и возвращает левый операнд – он устанавливает результат, потому что истина в операции `or` с чем угодно всегда дает истину. В остальных двух проверках левый операнд является ложным (пустой объект), так что Python просто продолжает оценку и возвращает правый операнд, который при проверке может дать либо истинное, либо ложное значение.

Оценка операции `and` в Python также прекращается, как только результат становится известным; однако в данном случае Python оценивает операнды слева направо и останавливается, если левый операнд оказывается ложным объектом, т.к. он определяет результат – ложь в операции `and` с чем угодно всегда дает ложь:

```
>>> 2 and 3, 3 and 2    # Возвращает левый операнд, если он ложный
(3, 2)                  # В противном случае возвращает правый операнд
                        # (истинный или ложный)
>>> [] and {}
[]
>>> 3 and []
[]
```

Здесь в первой строке оба операнда истинны, поэтому Python оценивает обе части операции и возвращает правый операнд. Во второй проверке левый операнд ложный ([]), так что Python останавливает оценку и возвращает его в качестве результата. В последней проверке левая сторона операции является истинной (3), а потому Python продолжает оценку и возвращает правый операнд, который оказывается ложным ([]).

Конечный результат всех проверок такой же, как в С и большинстве других языков – вы получаете значение, которое является логически истинным или ложным при проверке в `if` либо `while` согласно нормальным определениям `or` и `and`. Тем не менее, булевские операции в Python возвращают *объект* левого или правого операнда, а не простой целочисленный признак.

Подобное поведение `and` и `or` поначалу может показаться экзотическим, но во врезке “Что потребует внимания: булевские операции” далее в главе приводятся примеры его применения при программировании на Python. В следующем разделе показан распространенный способ использования такого поведения в своих интересах и его более наглядная замена в последних версиях Python.

Тернарное выражение `if/else`

Булевские операции, описанные в предыдущем разделе, часто применяются для написания выражений, выполняемых точно так же, как операторы `if`. Взгляните на показанный ниже оператор, который устанавливает A либо в Y, либо в Z на основе значения истинности X:

```
if X:  
    A = Y  
else:  
    A = Z
```

Однако временами элементы, задействованные в таком операторе, настолько просты, что их распространение на четыре строки выглядит излишеством. В других случаях конструкцию подобного рода может понадобиться вложить в более крупный оператор, а не присваивать ее результат какой-то переменной. По указанным причинам (и, откровенно говоря, из-за наличия похожего инструмента в языке C) в версии Python 2.5 появился новый формат выражения, который позволяет определить то же самое в одном выражении:

```
A = Y if X else Z
```

Приведенное выражение дает в точности такой же результат, что и предшествующий четырехстрочный оператор `if`, но проще в написании. Как и в эквиваленте в виде оператора, Python выполняет выражение `Y`, только если `X` оказывается истинным, а выражение `Z` — только если `X` оказывается ложным. То есть Python производит укороченную оценку, как и в булевых операциях, которые обсуждались в предыдущем разделе, выполняя только выражение `Y` или `Z`, но не оба. Рассмотрим несколько примеров:

```
>>> A = 't' if 'spam' else 'f'      # Непустые строки означают истину  
>>> A  
't'  
>>> A = 't' if '' else 'f'  
>>> A  
'f'
```

До выхода версии Python 2.5 (и после Python 2.5, если уж на то пошло) того же эффекта часто можно было достичь, аккуратно комбинируя операции `and` и `or`, потому что они возвращают либо левый, либо правый объект, как было описано в предыдущем разделе:

```
A = ((X and Y) or Z)
```

Код работает, но есть одна загвоздка — вы должны быть в состоянии предположить, что `Y` будет булевской истиной. В таком случае эффект оказывается тем же: операция `and` выполняется первой и возвращает `Y`, если `X` истинно; если `X` ложно, тогда и пропускает `Y`, а операция `or` просто возвращает `Z`. Другими словами, мы получаем вариант “если `X`, тогда `Y`, иначе `Z`”. А вот эквивалент в тернарной форме:

```
A = Y if X else Z
```

Понимание формы с комбинацией `and/or` вероятно потребует “момента прозрения”, особенно когда она встречается в первый раз. Начиная с версии Python 2.5, в этой форме больше нет необходимости — когда нужна такая структура, используйте эквивалентное более надежное и ясное выражение `if/else` или полный оператор `if`, если части конструкции нетривиальны.

В качестве стороннего замечания. Применение следующего выражения в Python дает аналогичный результат, поскольку функция `bool` будет транслировать `X` в эквивалентное целое число 1 или 0, которое затем можно использовать как смещение для выбора истинных и ложных значений из списка:

```
A = [Z, Y][bool(X)]
```

Например:

```
>>> ['f', 't'][bool('')]  
'f'  
>>> ['f', 't'][bool('spam')]  
't'
```

Тем не менее, это не в точности то же самое, потому что Python не будет предпринимать укороченную оценку — он всегда выполняет и Z, и Y независимо от значения X. Из-за таких сложностей в Python 2.5 и последующих версиях лучше применять более простое и легкое для восприятия выражение if/else. Однако опять-таки вы должны использовать его крайне умеренно и только когда все части относительно просты; иначе предпочтительнее записывать форму полного оператора if, чтобы облегчить внесение изменений в будущем. Ваши коллеги по работе будут счастливы.

И все же вы можете встречать версию с операциями and/or в коде, написанном до выхода версии Python 2.5 (и в коде Python, написанном бывшими программистами на C, которые еще не полностью отпустили свое темное прошлое кодирования)².

Что потребует внимания: булевские операции

Один из распространенных способов применения нескольких необычного поведения булевых операций Python — выбор из набора объектов с помощью or. Оператор следующего вида:

```
X = A or B or C or None
```

присваивает X первый непустой (т.е. истинный) объект из числа A, B и C или None, если все они пусты. Прием работает из-за того, что операция or возвращает один из двух своих объектов, и представляет собой довольно часто используемую кодовую парадигму в Python: для выбора непустого объекта из набора с фиксированным размером необходимо объединить их вместе в выражении or. В более простой форме это также применяется для указания стандартного значения — показанный ниже код устанавливает X в A, если A является истинным (или непустым), и в default в противном случае:

```
X = A or default
```

Также важно понимать укороченную оценку булевых операций и выражения if/else, потому что это может препятствовать выполнению действий. Скажем, выражение в правой части булевой операции может вызывать функции, которые делают значимую или важную работу, либо иметь побочные эффекты, которые не произойдут по причине вступления в силу правила укороченной оценки:

```
if f1() or f2(): ...
```

Здесь если f1 возвращает истинное (или непустое) значение, тогда Python никогда не запустит f2. Чтобы гарантировать выполнение обеих функций, их нужно вызывать перед операцией or:

```
tmp1, tmp2 = f1(), f2()  
if tmp1 or tmp2: ...
```

² На самом деле выражение Y if X else Z в Python имеет порядок, слегка отличающийся от выражения X ? Y : Z в языке C, и задействует более читабельные слова. Как сообщают, его отличающийся порядок был выбран в ответ на анализ широко употребляемых шаблонов в коде Python. В соответствии с фольклором такой порядок был выбран отчасти и для того, чтобы препятствовать его чрезмерному использованию бывшими программистами на C! Помните, простое лучше сложного, как в Python, так и в других местах. Если вам приходится упорно работать над упаковкой логики в выражения подобного рода, тогда операторы, возможно, будут более удачным вариантом.

Вы уже видели еще одно применение такого поведения ранее в главе: из-за способа работы булевых операций выражение `((A and B) or C)` может использоваться для эмуляции оператора `if` – почти полной.

Дополнительные сценарии применения булевых операций встречались в предшествующих главах. Как было показано в главе 9, поскольку все объекты по своей сути считаются истинными или ложными, в Python принято и более просто проверять объект напрямую (`if X:`), чем сравнивать его с пустым значением (`if X != ''`). Для строки обе проверки эквивалентны. В главе 5 объяснялось, что предварительно установленные булевые значения `True` и `False` являются тем же, что и целые числа 1 и 0; они пригодны для инициализации переменных (`X = False`), для проверок в циклах (`while True:`) и для отображения результатов в интерактивной подсказке.

Также дождитесь связанного обсуждения перегрузки операций в части VI: когда мы определяем новые типы объектов с помощью классов, то можем задавать их булевскую природу посредством методов `_bool_` или `_len_` (метод `_bool_` в Python 2.7 называется `_nonzero_`). Метод `_len_` вызывается при отсутствии метода `_bool_` и обозначает ложь, возвращая длину ноль – пустой объект трактуется как ложный.

Наконец, в качестве предварительного представления отметим, что в Python есть другие инструменты с ролями, похожими на цепочки `or` в начале данной врезки. Вызов `filter` и списковые включения, которые будут рассматриваться позже, могут использоваться для выбора истинных значений, когда набор кандидатов не известен вплоть до времени выполнения (хотя они оценивают все значения и возвращают все истинные из них). Встроенные функции `any` и `all` могут применяться для проверки, является любое или все элементы в коллекции истинными (правда, они не выбирают элемент):

```
>>> L = [1, 0, 2, 0, 'spam', '', 'ham', []]
>>> list(filter(bool, L))      # Получение истинных значений
[1, 2, 'spam', 'ham']
>>> [x for x in L if x]        # Включения
[1, 2, 'spam', 'ham']
>>> any(L), all(L)            # Накопление значений истинности
(True, False)
```

Как было показано в главе 9, функция `bool` просто возвращает истинное или ложное значение своего аргумента, как будто бы оно проверялось в `if`. Дополнительные сведения о таких связанных инструментах будут приведены в главах 14, 19 и 20.

Резюме

В главе мы исследовали оператор `if` языка Python. Кроме того, поскольку он был первым составным и логическим оператором, мы пересмотрели общие синтаксические правила Python и более глубоко проанализировали работу значений истинности и проверок, чем были в состоянии делать это до сих пор. Попутно мы также взглянули на способы кодирования множественного ветвления в Python, узнали о выражении `if/else`, появившемся в Python 2.5, и ознакомились с распространенными случаями появления булевых значений в коде.

В следующей главе мы продолжим рассмотрение процедурных операторов подробным обсуждением циклов `while` и `for`. Вы узнаете об альтернативных способах написания циклов в Python, часть которых может быть лучше других. Но прежде чем двигаться дальше, по традиции ответьте на контрольные вопросы главы.

Проверьте свои знания: контрольные вопросы

1. Как вы могли бы закодировать множественное ветвление в Python?
2. Как вы могли бы закодировать оператор `if/else` в форме выражения в Python?
3. Как вы могли бы распространить одиночный оператор на несколько строк?
4. Что означают слова `True` и `False`?

Проверьте свои знания: ответы

1. Наиболее прямолинейным способом кодирования множественного ветвления часто оказывается оператор `if` с множеством конструкций `elif`, хотя он не обязательно будет самым лаконичным или гибким. Того же результата нередко можно достичь с помощью индексации словаря, особенно если словарь содержит вызываемые функции, реализованные посредством операторов `def` или выражений `lambda`.
2. В Python 2.5 и последующих версиях форма выражения `Y if X else Z` возвращает `Y`, если `X` истинно, или `Z` в противном случае; она эквивалентна четырехстрочному оператору `if`. Комбинация `and/or (((X and Y) or Z))` может работать аналогично, но менее ясна и требует, чтобы часть `Y` была истинной.
3. Помещение оператора в синтаксическую пару скобок `((), []` или `{ }`) дает возможность разнести его на любое желаемое количество строк; оператор заканчивается, когда Python встречает закрывающую (правую) скобку из пары, а строки 2 и далее оператора могут начинаться с любым уровнем отступа. Продолжение с помощью обратной косой черты тоже работает, но крайне не одобряется в мире Python.
4. `True` и `False` — всего лишь специальные версии целых чисел 1 и 0: в Python они всегда обозначают булевские значения истины и лжи. Они доступны для использования при проверке истинности и инициализации переменных, а также выводятся для результатов выражений в интерактивной подсказке. Во всех упомянутых ролях они выступают в качестве более значащей и оттого читабельной альтернативы 1 и 0.

Циклы `while` и `for`

Настоящей главой мы завершаем наш тур по процедурным операторам Python представлением двух главных циклических конструкций языка – операторов, которые повторяют действие снова и снова. Первый из них, оператор `while`, предлагает способ написания универсальных циклов. Второй, оператор `for`, предназначен для прохода по элементам в последовательности или в другом итерируемом объекте и выполнения блока кода для каждого элемента.

Неформально мы уже видели оба оператора, но здесь будут восполнены недостающие детали, касающиеся использования. По ходу дела мы также исследуем менее заметные операторы, применяемые внутри циклов, такие как `break` и `continue`, а также раскроем ряд встроенных функций, часто используемых с циклами, в том числе `range`, `zip` и `map`.

Хотя рассматриваемые в главе операторы `while` и `for` представляют собой главный синтаксис, предлагаемый для кодирования повторяющихся действий, в Python существуют другие процессы и концепции организации циклов. В связи с этим история об итерации продолжается в следующей главе, где мы будем исследовать связанные идеи *протокола итерации* Python (применимого циклом `for`) и *списковых включений* (близкий родственник цикла `for`). В более поздних главах обсуждаются даже более экзотические инструменты итерации, такие как *генераторы*, `filter` и `reduce`. Но пока давайте займемся простыми вещами.

Циклы `while`

Оператор `while` – самая универсальная конструкция для итераций в языке Python. Выражаясь простыми терминами, он многократно выполняет блок операторов (обычно с отступом) до тех пор, пока проверка в заголовочной части оценивается как истинное значение. Это называется “циклом”, потому что управление продолжает возвращаться к началу оператора, пока проверка не даст ложное значение. Когда результат проверки становится ложным, управление переходит на оператор, следующий после блока `while`. Совокупный эффект в том, что тело цикла выполняется многократно, пока проверка в заголовочной части дает истинное значение. Если проверка оценивается в ложное значение с самого начала, тогда тело цикла никогда не выполнится и оператор `while` пропускается.

Общий формат

В своей самой сложной форме оператор `while` состоит из строки заголовка с выражением проверки, тела с одним или большим количеством оператором с отступами

и необязательной части `else`, которая выполняется, если управление покидает цикл, а оператор `break` не встретился. Python продолжает оценивать выражение проверки в строке заголовка и выполняет операторы, вложенные в тело цикла, пока проверка не возвратит ложное значение:

```
while проверка:      # Проверка цикла
    операторы        # Тело цикла
else:                # Необязательная часть else
    операторы        # Выполняются, если не произведен выход из цикла с помощью break
```

Примеры

В целях иллюстрации давайте посмотрим на несколько простых циклов `while` в действии. Первый, который состоит из оператора `print`, вложенного в цикл `while`, всего лишь бесконечно выводит сообщение. Вспомните, что `True` – это специальная версия целого числа `1` и всегда обозначает булевское истинное значение; поскольку проверка всегда дает истину, Python продолжает выполнение тела до бесконечности или до тех пор, пока вы не остановите его выполнение. Поведение такого вида обычно называется **бесконечным циклом** – по правде говоря, он не вечен, но вам может понадобиться нажать комбинацию клавиш `<Ctrl+C>`, чтобы принудительно закончить его работу:

```
>>> while True:
...     print('Type Ctrl-C to stop me!')
```

В следующем примере производится отбрасывание первого символа строки до тех пор, пока она не станет пустой и потому ложной. Вполне обычно проверять объект напрямую вместо использования более многословного эквивалента (`while x != ''`). Позже в главе мы рассмотрим другие способы прохода по элементам в строке, которые более легко реализуются посредством цикла `for`.

```
>>> x = 'spam'
>>> while x:                      # До тех пор, пока строка x не пустая
...     print(x, end=' ')           # В Python 2.X: print x,
...     x = x[1:]                   # Отбрасывание первого символа строки x
...
spam pam am m
```

Обратите внимание на ключевой аргумент `end=' '`, применяемый здесь для размещения всех выводов в той же строке с разделением их пробелами; обратитесь в главу 11, если забыли, почему это работает именно так. В конце вывода приглашение на ввод может остаться в странном состоянии; введите `<Enter>`, чтобы сбросить его. Пользователи Python 2.X также должны помнить об использовании хвостовой запятой вместо ключевого аргумента `end` в конструкциях `print` такого рода.

В показанном ниже коде производится подсчет от значения `a` до значения `b`, не включая его. Позже мы увидим более легкий способ такого подсчета с помощью цикла `for` языка Python и встроенной функции `range`:

```
>>> a=0; b=10
>>> while a < b:                  # Один способ написания циклов с подсчетом
...     print(a, end=' ')
...     a += 1                         # Или a = a + 1
...
0 1 2 3 4 5 6 7 8 9
```

Обратите внимание, что Python не располагает тем, что в других языках называется оператором цикла “do until”. Однако мы можем эмулировать его посредством проверки и оператора `break` в конце тела цикла, так что тело цикла всегда выполняется хотя бы раз:

```
while True:  
    ...цело цикла...  
    if exitTest(): break
```

Чтобы полностью понять, как работает такая структура, нам необходимо перейти в следующий раздел и больше узнать об операторе `break`.

Операторы `break`, `continue`, `pass` и конструкция `else` цикла

Теперь, когда было продемонстрировано несколько циклов Python в действии, наступило время взглянуть на два простых оператора, которые достигают своих целей, только когда вложены внутри циклов – `break` и `continue`. Занимаясь здесь такими необычными операторами, мы также рассмотрим конструкцию `else` цикла, потому что она переплетена с `break`, и пустой оператор-заполнитель `pass` (который не привязан к циклам как таковой, но относится к общей категории простых однословных операторов). Указанные инструменты Python кратко описаны ниже.

<code>break</code>	Переходит за пределы ближайшего заключающего цикла (после всего оператора цикла).
<code>continue</code>	Переходит в начало ближайшего заключающего цикла (на строку заголовка цикла).
<code>pass</code>	Вообще ничего не делает: это пустой оператор-заполнитель.
Блок <code>else</code> цикла	Выполняется тогда и только тогда, когда происходит нормальный выход из цикла (т.е. без выполнения оператора <code>break</code>).

Общий формат цикла

С учетом операторов `break` и `continue` общий формат цикла `while` выглядит следующим образом:

```
while проверка:  
    операторы  
    if проверка: break      # Выход из цикла с пропуском else, если есть  
    if проверка: continue   # Переход на проверку в начале цикла  
else:  
    операторы              # Выполняется, если не было break
```

Операторы `break` и `continue` могут появляться где угодно внутри тела цикла `while` (или `for`), но они обычно записываются с дополнительным отступом в операторе `if` для выполнения действия в ответ на некоторое условие.

Давайте перейдем к нескольким простым примерам, чтобы выяснить, как все эти операторы применяются вместе на практике.

Оператор `pass`

Начнем с простого: оператор `pass` – это заполнитель, обозначающий отсутствие действий, который используется в ситуациях, когда синтаксис требует оператора, но нет возможности выполнить что-либо полезное. Он часто применяется при кодиро-

вании пустого тела для составного оператора. Скажем, вот как с помощью `pass` написать бесконечный цикл, который на каждом проходе ничего не делает:

```
while True: pass      # Для прекращения работы нажмите <Ctrl+C>!
```

Поскольку тело представляет собой всего лишь пустой оператор, Python застрянет в таком цикле. `pass` в рамках операторов — примерно то же самое, что и `None` в рамках объектов, т.е. явное отсутствие чего-либо. Обратите внимание, что тело цикла `while` находится в одной строке с заголовком после двоеточия; как и в случае операторов `if`, поступать так разрешено, только если тело не является составным оператором.

Приведенный пример бездействует все время. Вряд ли его можно считать самой полезной программой Python из числа когда-либо написанных (если только вы не хотите разогреть свой ноутбук холодным зимним утром!); но откровенно говоря, мне не удалось придумать лучший пример использования `pass` в текущем месте книги.

Позже вы увидите случаи более осмысленного применения `pass` — например, для игнорирования исключений, перехватываемых операторами `try`, а также для определения пустых объектов `class` с атрибутами, которые ведут себя подобно “структурам” и “записям” в других языках. Иногда оператор `pass` обозначает место, подлежащее заполнению в будущем, что служит временной заглушкой для тел функций:

```
def func1():
    pass      # Позже поместить сюда реальный код
def func2():
    pass
```

Мы не можем оставить тело функции пустым, не получив синтаксической ошибки, поэтому взамен используем `pass`.



Примечание, касающееся нестыковки версий. В Python 3.X (но не Python 2.X) в любом месте, где может находиться выражение, разрешено указывать *многоточие*, которое записывается как ... (просто три следующие друг за другом точки). Поскольку сами по себе многоточия ничего не делают, они могут служить альтернативой оператору `pass`, особенно для кода, подлежащего заполнению в будущем:

```
def func1():
    ...
def func2():
    ...
func1()           # При вызове ничего не делает
```

Многоточия могут также располагаться в одной строке с заголовком оператора и применяться для инициализации имен переменных, если не требуется какой-то особый тип:

```
def func1(): ...      # Работает и в той же самой строке
def func2(): ...
>>> X = ...          # Альтернатива None
>>> X
Ellipsis
```

Форма записи подобного рода появилась в Python 3.X и выходит далеко за рамки первоначального предназначения ... для расширений срезов, поэтому время покажет, станет ли она достаточно распространенной для того, чтобы бросить вызов `pass` и `None` в таких ролях.

Оператор `continue`

Оператор `continue` вызывает немедленный переход в начало цикла. Иногда он также позволяет избежать вложения операторов. В приведенном далее примере оператор `continue` используется для пропуска вывода нечетных чисел. Код выводит все четные числа, которые меньше 10 и больше или равны 0. Вспомните, что 0 означает ложь, а `%` – операцию получения остатка от деления (модуля), поэтому данный цикл делает обратный отсчет до 0, пропуская числа, которые не являются множителями 2 – он выводит 8 6 4 2 0:

```
x = 10
while x:
    x = x-1
    if x % 2 != 0: continue      # Либо x -= 1
    print(x, end=' ')
```

Из-за того, что `continue` инициирует переход в начало цикла, нет необходимости вкладывать `print` внутрь проверки `if`; оператор `print` достигается, только если не будет выполнен `continue`. Если это выглядит похожим на “`goto`” в других языках, то так и есть. В Python оператор, подобный “`goto`”, отсутствует, но поскольку оператор `continue` позволяет совершать переходы в программе, то к нему применимы многие предупреждения относительно читабельности и удобства сопровождения, которые вы могли слышать о “`goto`”. Вероятно, оператор `continue` должен использоваться умеренно, особенно когда вы лишь начинаете программировать на Python. Скажем, последний пример может стать яснее, если `print` вложить в `if`:

```
x = 10
while x:
    x = x-1
    if x % 2 == 0:              # Четное? Тогда вывести
        print(x, end=' ')
```

Позже в книге вы узнаете, что с помощью генерации и перехвата исключений также можно эмулировать операторы “`goto`” ограниченными и структуризованными путями. Данная методика будет более подробно рассматриваться в главе 36, где мы обсудим ее применение для выхода из множества вложенных циклов, что невозможно сделать посредством одного лишь оператора `break`, рассматриваемого в следующем разделе.

Оператор `break`

Оператор `break` вызывает немедленный выход из цикла. Поскольку при его достижении код, который находится за ним в теле цикла, не выполняется, за счет включения `break` иногда можно избежать вложения. Например, ниже представлен простой интерактивный цикл (вариант более крупного примера, который приводился в главе 10), где с помощью `input` (`raw_input` в Python 2.X) вводятся данные; когда пользователь в ответ на запрос имени вводит слово `stop`, происходит выход из цикла:

```
>>> while True:
...     name = input('Enter name:')  # Использовать raw_input() в Python 2.X
...     if name == 'stop': break
...     age = input('Enter age: ')
...     print('Hello', name, '=>', int(age) ** 2)
... 
```

```
Enter name:bob
Enter age: 40
Hello bob => 1600
Enter name:sue
Enter age: 30
Hello sue => 900
Enter name:stop
```

Обратите внимание на преобразование введенного значения `age` в целое число посредством функции `int` перед его возведением в квадрат; вспомните, что поступать так необходимо из-за того, что `input` возвращает введенное пользователем значение в виде строки. В главе 36 вы увидите, что `input` также генерирует исключение при обнаружении признака конца файла (например, когда пользователь нажимает `<Ctrl+Z>` в Windows или `<Ctrl+D>` в Unix); если это важно, тогда поместите `input` внутрь оператора `try`.

Конструкция `else` цикла

В сочетании с конструкцией `else` цикла оператор `break` часто позволяет устраниить потребность во флагах состояния поиска, используемых в других языках. Скажем, следующий фрагмент кода определяет, является ли положительное целое число `y` простым, за счет поиска сомножителей больше 1:

```
x = y // 2                                # Для значений y > 1
while x > 1:
    if y % x == 0:                          # Остаток от деления
        print(y, 'has factor', x)           # Имеет сомножитель
        break                               # Пропуск else
    x -= 1
else:                                         # Нормальный выход
    print(y, 'is prime')                  # Является простым
```

Вместо установки флага, предназначенного для проверки, нужно ли выходить из цикла, в коде применяется оператор `break`, когда сомножитель найден. В итоге можно допустить, что конструкция `else` цикла будет выполняться, только если сомножители не найдены; попадание на `break` означает, что число простое. Выполните код пошагово, чтобы посмотреть, как он работает.

Конструкция `else` цикла также выполняется, если тело цикла ни разу не выполнилось, потому что в этом случае не выполняется и оператор `break`; в цикле `while` подобное происходит, когда проверка в заголовке с самого начала дает ложное значение. Таким образом, в предыдущем примере все равно будет выводиться сообщение `is prime`, если `x` изначально меньше или равно 1 (т.е. когда `y` равно 2).



В рассмотренном примере выявлялись простые числа, но лишь неформально. Согласно строгому математическому определению числа меньше 2 не считаются простыми. Более того, такой код также терпит неудачу для отрицательных чисел и успешно выполняется для чисел с плавающей точкой без дробной части. Вдобавок следует отметить, что в случае Python 3.X в коде вместо операции `/` должна использоваться операция `//` из-за превращения `/` в “настоящее деление”, как было описано в главе 5 (начальное деление необходимо для усечения остатка, а не для его предохранения!). Если вы хотите поэкспериментировать с данным кодом, тогда обратитесь к упражнению в конце части IV, где он помещается внутрь функции для многократного применения.

Дополнительные сведения о конструкции `else` цикла

Поскольку конструкция `else` цикла присутствует только в Python, она может сбивать с толку новичков (и не использоваться некоторыми ветеранами; мне встречались те, кто даже не знал, что в циклах *имеется* `else!`). В общих чертах конструкция `else` цикла просто предлагает явный синтаксис для распространенного кодового сценария — она представляет собой кодовую структуру, которая позволяет перехватить “другой” выход из цикла, не устанавливая и не проверяя флаги или условия.

Предположим, например, что мы пишем цикл для поиска значения в списке и хотим знать, было ли значение найдено, после выхода из цикла. Решить задачу можно было бы следующим образом (код намеренно оставлен абстрактным и незавершенным; `x` — это последовательность, а `match` — функция проверки, которая должна быть определена):

```
found = False
while x and not found:
    if match(x[0]):           # Значение в начале?
        print('Ni')
        found = True
    else:
        x = x[1:]            # Усечь в начале и повторить
if not found:
    print('not found')      # Значение не найдено
```

Здесь мы инициализируем, устанавливаем и позже проверяем флаг для определения, успешным ли был поиск. Это допустимый код Python, и он работает; тем не менее, он точно отражает разновидность структуры, для поддержки которой предназначена конструкция `else` цикла. Вот эквивалентный код с конструкцией `else`:

```
while x:                      # Выйти, когда x пусто
    if match(x[0]):            # Проверка
        print('Ni')
        break                  # Выйти, пропустить else
    x = x[1:]
else:
    print('Not found')       # Выполняется только в случае исчерпания x
```

Версия с конструкцией `else` цикла лаконичнее. Флаг исчез, а проверка `if` после конца цикла заменена конструкцией `else` (вертикально выровненной со словом `while`). Поскольку `break` внутри главной части `while` производит выход из цикла и пропускает `else`, мы имеем более структурированный способ перехвата случая с неудавшимся поиском.

Некоторые читатели могли заметить, что конструкцию `else` в предыдущем примере можно было бы заменить проверкой, пуст ли список `x`, после цикла (`if not x:`). Хотя это справедливо в рассмотренном примере, конструкция `else` предлагает явный синтаксис для такого кодового шаблона (она здесь является очевидной конструкцией для случая с неудавшимся поиском), а явная проверка, пуст ли список, в определенных ситуациях может оказаться неприменимой. Конструкция `else` цикла становится даже более удобной, когда используется в сочетании с циклом `for` (тема следующего раздела), потому что итерация по последовательности не находится под вашим контролем.

Что потребует внимания: эмуляция циклов `while` языка С

В разделе главы 11, посвященном операторам выражений, было указано, что Python не разрешает операторам вроде присваивания находиться в местах, где ожидается выражение. То есть каждый такой оператор в общем случае должен располагаться в отдельной строке, а не быть вложенным в более крупную конструкцию. Это означает, что распространенный кодовый шаблон языка C в Python работать не будет:

```
while ((x = next(obj)) != NULL) {...обработка x...}
```

Присваивания в C возвращают присвоенные значения, но присваивания в Python являются просто операторами, а не выражениями. Тем самым устраняется пользующаяся дурной славой категория ошибок в C: вы не можете в Python случайно набрать `=`, когда имели в виду `==`. Однако если вам необходимо похожее поведение, то есть по меньшей мере три способа достичь того же эффекта в циклах `while` языка Python, не встраивая присваивания в проверки циклов. Вы можете перенести присваивание в тело цикла и добавить `break`:

```
while True:  
    x = next(obj)  
    if not x: break  
    ...обработка x...
```

или перенести присваивание в цикл с проверками:

```
x = True  
while x:  
    x = next(obj)  
    if x:  
        ...обработка x...
```

либо вынести первое присваивание за пределы цикла:

```
x = next(obj)  
while x:  
    ...обработка x...  
    x = next(obj)
```

Первый кодовый шаблон из показанных трех некоторые могут посчитать наименее структурированным, но он также проще и применяется чаще. Простой цикл `for` языка Python также может заменить подобные циклы C, но в языке C его прямой аналог отсутствует:

```
for x in obj: ...обработка x...
```

Циклы `for`

Цикл `for` является универсальным итератором в Python: он может проходить по элементам в любой упорядоченной последовательности или в другом итерируемом объекте. Оператор `for` работает на строках, списках, кортежах и прочих встроенных итерируемых объектах, а также на новых объектах, определяемых пользователем, которые мы позже научимся создавать с помощью классов. Мы кратко касались `for` в главе 4 и в сочетании с типами объектов последовательностей; давайте рассмотрим его использование более формально.

Общий формат

Цикл `for` языка Python начинается со строки заголовка, где указывается цель (или цели) присваивания наряду с объектом, по которому нужно совершить проход. После заголовка находится блок операторов (обычно с отступами), который необходимо повторять:

```
for цель in объект:  
    операторы  
else:  
    операторы
```

Присваивает цели элементы объекта
Повторяющее тело цикла: использует цель
Необязательная часть else
Если не встречался оператор break

Когда Python запускает цикл `for`, он присваивает цели элементы итерируемого объекта по очереди и выполняет для каждого тела цикла. Внутри тела цикла цель присваивания обычно используется для ссылки на текущий элемент в последовательности, как если бы цель была курсором, проходящим через последовательность.

Имя, применяемое как цель присваивания в строке заголовка `for`, обычно является (возможно, новой) переменной внутри области видимости, где находится оператор `for`. Имя не отличается какой-то уникальностью; его даже можно изменять внутри тела цикла, но оно будет автоматически устанавливаться в следующий элемент последовательности, когда управление снова возвратится в начало цикла. После цикла эта переменная, как правило, по-прежнему ссылается на последний посещенный элемент, которым будет последний элемент в последовательности, если только не произошел выход из цикла посредством оператора `break`.

Оператор `for` также поддерживает необязательный блок `else`, который работает точно как в цикле `while` – он выполняется, если выход из цикла осуществляется без помощи оператора `break` (т.е. когда были посещены все элементы последовательности). Представленные ранее операторы `break` и `continue` в цикле `for` также работают аналогично циклу `while`. Полный формат цикла `for` может быть описан следующим образом:

```
for цель in объект:  
    операторы  
    if проверка: break  
    if проверка: continue  
else:  
    операторы
```

Присваивает цели элементы объекта
Выход из цикла с пропуском else
Переход в начало цикла
Если не встречался оператор break

Примеры

Давайте теперь наберем несколько циклов `for` в интерактивной подсказке, чтобы посмотреть, каким образом они используются на практике.

Базовое использование

Как упоминалось ранее, цикл `for` может проходить по объекту последовательности любого вида. В нашем первом примере мы будем присваивать имени `x` каждый из трех элементов по очереди, слева направо, и для каждого из них будет выполняться оператор `print`. Внутри оператора `print` (тело цикла) имя `x` ссылается на текущий элемент в списке:

```
>>> for x in ["spam", "eggs", "ham"]:  
...     print(x, end=' ')  
...  
spam eggs ham
```

В следующих двух примерах вычисляется сумма и произведение всех элементов в списке. Позже в главе и книге будут встречаться инструменты, которые автоматически применяют операции вроде + и * к элементам списка, но использовать for зачастую не сложнее:

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item
...
>>> prod
24
```

Другие типы данных

В цикле for работает любая последовательность, т.к. он представляет собой универсальный инструмент. Скажем, циклы for работают на строках и кортежах:

```
>>> S = "lumberjack"
>>> T = ("and", "I'm", "okay")

>>> for x in S: print(x, end=' ')           # Итерация по строке
...
l u m b e r j a c k

>>> for x in T: print(x, end=' ')           # Итерация по кортежу
...
and I'm okay
```

На самом деле, как вы узнаете в следующей главе при исследовании понятия “итерируемых объектов”, циклы for способны даже работать на некоторых объектах, не являющихся последовательностями – например, файлах и словарях.

Присваивание кортежей в циклах for

При проходе по последовательности кортежей сама переменная цикла фактически будет *кортежем целей*. Это всего лишь еще один случай распаковывающего присваивания кортежей, работа которого демонстрировалась в главе 11. Вспомните, что цикл for присваивает цели элементы в объекте последовательности и присваивание работает одинаково повсюду:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:                  # Присваивание кортежей в действии
...     print(a, b)
...
1 2
3 4
5 6
```

Первый проход цикла подобен написанию $(a, b) = (1, 2)$, второй проход – $(a, b) = (3, 4)$ и т.д. Совокупным эффектом оказывается автоматическая распаковка текущего кортежа на каждой итерации.

Такая форма обычно применяется в сочетании с вызовом встроенной функции `zip`, которую мы увидим позже в главе при реализации параллельных обходов. Она также регулярно встречается вместе с базами данных SQL в Python, когда таблицы результатов запроса возвращаются в виде последовательностей, подобных использованному здесь списку — внешний список является таблицей базы данных, вложенные кортежи представляют собой строки внутри таблицы, а присваивание кортежей извлекает столбцы.

Кортежи в циклах `for` также становятся полезными при итерации *сразу* по ключам и значениям в словарях с применением метода `items` вместо прохода в цикле по ключам и индексации для извлечения значений вручную:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for key in D:
...     print(key, '=>', D[key])      # Использование итерации по ключам
...                                     # словаря и индексации
...
a => 1
c => 3
b => 2

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (key, value) in D.items():
...     print(key, '=>', value)      # Итерация сразу по ключам и значениям
...
a => 1
c => 3
b => 2
```

Важно отметить, что присваивание кортежей в циклах `for` — это не особый случай; после слова `for` синтаксически допускается любая цель присваивания. Для распаковки мы всегда можем присваивать вручную внутри цикла:

```
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
...     a, b = both    # Эквивалент в виде присваивания вручную
...     print(a, b)    # Python 2.X: print с включающими круглыми скобками для кортежа
...
1 2
3 4
5 6
```

Но кортежи в заголовке цикла экономят добавочный шаг при итерации по последовательности последовательностей. Как было показано в главе 11, в цикле `for` подобным образом могут автоматически распаковываться также и *вложенные структуры данных*:

```
>>> ((a, b), c) = ((1, 2), 3)  # Работает также и для вложенных последовательностей
>>> a, b, c
(1, 2, 3)
>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)
...
1 2 3
4 5 6
```

Тем не менее, даже это не является особым случаем – цикл `for` просто выполняет на каждой итерации разновидность присваивания, которая делалась прямо перед ним. Таким способом может быть распакована любая структура с вложенными последовательностями просто из-за настолько универсальной природы *присваивания последовательностей*:

```
>>> for ((a, b), c) in([(1, 2), 3), ['XY', 6]): print(a, b, c)
...
1 2 3
X Y 6
```

Расширенное присваивание последовательностей в циклах `for` в Python 3.X

На самом деле, поскольку переменной цикла в `for` может быть любая цель присваивания, мы также можем использовать здесь расширенный синтаксис присваивания последовательностей с распаковкой Python 3.X для извлечения элементов и сегментов последовательностей внутри последовательностей. Действительно, как обсуждалось в главе 11, это тоже не особый случай, а просто новая форма присваивания в Python 3.X; поскольку он работает в операторах присваивания, то автоматически срабатывает в циклах `for`.

Рассмотрим присваивание кортежей, представленное в предыдущем разделе. На каждой итерации кортежу имен присваивается кортеж значений, в точности подобно простому оператору присваивания:

```
>>> a, b, c = (1, 2, 3)                                # Присваивание кортежей
>>> a, b, c
(1, 2, 3)

>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: # Используется в цикле for
...     print(a, b, c)
...
1 2 3
4 5 6
```

Из-за того, что в Python 3.X последовательность может быть присвоена более общему набору имен с помеченным звездочкой именем для сбора множества элементов, тот же самый синтаксис может применяться для извлечения частей вложенных последовательностей в цикле `for`:

```
>>> a, *b, c = (1, 2, 3, 4)    # Расширенное присваивание последовательностей
>>> a, b, c
(1, [2, 3], 4)

>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]: 
...     print(a, b, c)
...
1 [2, 3] 4
5 [6, 7] 8
```

На практике такой подход можно было бы использовать для выбора множества столбцов из строк данных, представленных как вложенные последовательности. В Python 2.X имена со звездочкой не разрешены, но похожего эффекта можно достичь нарезанием. Единственное отличие в том, что нарезание возвращает специфичный к типу результат, тогда как помеченным звездочкой именам всегда присваиваются списки:

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]: # Нарезание вручную в Python 2.X
...     a, b, c = all[0], all[1:3], all[3]
...     print(a, b, c)
...
1 (2, 3) 4
5 (6, 7) 8
```

Более подробно эта форма записи обсуждалась в главе 11.

Вложенные циклы `for`

Давайте теперь взглянем на цикл `for`, который сложнее тех, что мы видели до сих пор. В приведенном ниже примере иллюстрируется вложение операторов и конструкция `else` цикла `for`. Имея список объектов (`items`) и список ключей (`tests`), код ищет каждый ключ в списке объектов и сообщает о результате поиска:

```
>>> items = ["aaa", 111, (4, 5), 2.01]           # Набор объектов
>>> tests = [(4, 5), 3.14]                      # Ключи для поиска
>>>
>>> for key in tests:                           # Для всех ключей
...     for item in items:                        # Для всех элементов
...         if item == key:                         # Проверка на совпадение
...             print(key, "was found")            # Ключ найден
...             break
...     else:
...         print(key, "not found!")            # Ключ не найден
...
(4, 5) was found
3.14 not found!
```

Поскольку во вложенном операторе `if` выполняется `break`, когда совпадение обнаружено, в конструкции `else` цикла можно предположить, что если она достигнута, то поиск потерпел неудачу. Обратите внимание на вложение. После запуска кода одновременно выполняются два цикла: внешний цикл просматривает список ключей, а внутренний — список элементов для каждого ключа. Вложение конструкции `else` цикла критически важно; она смешена на тот же самый уровень, что и строка заголовка внутреннего цикла `for`, поэтому ассоциируется с внутренним циклом, но не с оператором `if` или внешним циклом `for`.

Пример является иллюстративным, но его код можно упростить, если использовать операцию `in` для проверки членства. Из-за того, что операция `in` неявно просматривает объект в поиске совпадения (по крайней мере, логически), она заменяет внутренний цикл:

```
>>> for key in tests:                           # Для всех ключей
...     if key in items:    # Позволить Python проверять на предмет совпадения
...         print(key, "was found")            # Ключ найден
...     else:
...         print(key, "not found!")          # Ключ не найден
...
(4, 5) was found
3.14 not found!
```

В целом ради краткости и высокой производительности разумно поручать Python выполнение как можно большего объема работы (что иллюстрируется в показанном выше решении).

Представленный далее пример похож, но вместо вывода в нем строится список для использования в будущем. Он выполняет посредством `for` типичную задачу со структурами данных – сбор элементов, общих в двух последовательностях (строках), – и служит грубой реализацией процедуры пересечения множеств. После выполнения цикла `res` ссылается на список, который содержит все элементы, находящиеся и в `seq1`, и в `seq2`:

```
>>> seq1 = "spam"
>>> seq2 = "scam"
>>>
>>> res = []                      # Начать с пустого списка
>>> for x in seq1:                 # Просмотр первой последовательности
...     if x in seq2:               # Общий элемент?
...         res.append(x)           # Добавить в конец результирующего списка
...
>>> res
['s', 'a', 'm']
```

К сожалению, написанный код способен работать только с двумя специфическими переменными: `seq1` и `seq2`. Было бы неплохо обобщить этот цикл, создав инструмент, который позволил бы применять его более одного раза. Вы увидите, что такая простая идея приводит нас к [функциям](#), рассматриваемым в следующей части книги.

Код также демонстрирует классический шаблон [ск립тового включения](#) – сбор списка результатов с помощью итерации и необязательной фильтрующей проверки – и также может быть записан более лаконично:

```
>>> [x for x in seq1 if x in seq2]    # Позволить Python собрать результаты
['s', 'a', 'm']
```

Но остаток истории будет описан в следующей главе.

Что потребует внимания: приемы просмотра файлов

Вообще говоря, циклы оказываются полезными везде, где необходимо повторять операцию или обработку чего-либо более одного раза. Поскольку *файлы* содержат множество символов и строк, они считаются одним из наиболее типичных сценариев использования для циклов. Для загрузки сразу всего содержимого файла в строку понадобится просто вызвать метод `read` файлового объекта:

```
file = open('test.txt', 'r')          # Чтение содержимого в строку
print(file.read())
```

Но для загрузки файла меньшими порциями обычно пишется либо цикл `while` с выходом при обнаружении конца файла, либо цикл `for`. Для [посимвольного чтения](#) будет достаточно любого из показанных ниже двух циклов:

```
file = open('test.txt')
while True:
    char = file.read(1)              # Посимвольное чтение
    if not char: break              # Пустая строка означает конец файла
    print(char)

for char in open('test.txt').read():
    print(char)
```

Цикл `for` здесь также обрабатывает каждый символ, но файл загружается в память целиком (и предполагается, что он умещается в памяти!). Чтобы заменить читать по *строкам* или *блокам*, можно применять цикл `while` такого вида:

```
file = open('test.txt')
while True:
    line = file.readline()          # Чтение строки за строкой
    if not line: break
    print(line.rstrip())           # Стока уже содержит \n

file = open('test.txt', 'rb')
while True:
    chunk = file.read(10)          # Чтение байтовых порций: до 10 байтов
    if not chunk: break
    print(chunk)
```

Байтовые данные обычно читаются по блокам. Однако чтобы читать текстовые файлы *строка за строкой*, легче написать цикл `for`, который еще и будет быстрее выполнятся:

```
for line in open('test.txt').readlines():
    print(line.rstrip())

for line in open('test.txt'):
    # Использование итераторов:
    # лучше для текстового ввода
    print(line.rstrip())
```

Обе версии работают в Python 2.X и 3.X. В первой используется файловый метод `readlines` для загрузки всего содержимого файла в список строк, а вторая версия полагается на файловые *итераторы* для автоматического чтения по одной строке на каждой итерации цикла.

Последний пример также в целом будет *наилучшим* вариантом для текстовых файлов — помимо простоты он способен работать с произвольно крупными файлами, потому что не загружает в память сразу все содержимое файла. Кроме того, версия с итератором может оказаться более быстрой, хотя производительность ввода-вывода варьируется в зависимости от линейки и выпуска Python.

Тем не менее, вызовы файлового метода `readlines` по-прежнему могут быть полезны, скажем, для изменения на *противоположный* порядка следования строк из файла при условии, что его содержимое способно уместиться в памяти. Встроенная функция `reversed` принимает последовательность, но не произвольный итерируемый объект, который генерирует значения; другими словами, список подойдет, но файловый объект — нет:

```
for line in reversed(open('test.txt').readlines()): ...
```

В некотором коде Python 2.X вы можете заметить, что имя `open` заменено именем `file` и для достижения такого же результата, как с автоматическим итератором строк файла, применяется более старый метод `xreadlines` файлового объекта (он похож на `readlines`, но не загружает в память сразу весь файл). В версии Python 3.X как `file`, так и `xreadlines` были удалены, поскольку они избыточны. В целом при написании нового кода Python 2.X вы должны избегать их использования (в недавних выпусках Python 2.X применяйте файловые итераторы и вызов `open`), но они могут встречаться в более старом коде и ресурсах.

Дополнительные сведения о задействованных здесь вызовах ищите в руководство по библиотеке, а о файловых итераторах строк — в главе 14. Кроме того, почитайте врезку “Что потребует внимания: команды оболочки и другое” далее в главе; там те же самые файловые инструменты применяются к запускающему модулю `os.ropen` для чтения вывода программ. Более подробно о чтении файлов также рассказывается в главе 37; вы увидите, что в Python 3.X текстовые и двоичные файлы имеют несколько отличающуюся семантику.

Методики написания циклов

Только что исследованный цикл `for` относится к самой большой категории циклов с подсчетом. Он обычно проще в написании и часто выполняется быстрее, чем `while`, поэтому является первым инструментом, к которому вы должны обращаться всякий раз, когда необходимо проходить через последовательность или другой итерируемый объект. На самом деле, как правило, вы должны *противиться искушению подсчитывать что-либо в Python* – его итерационные средства автоматизируют большую часть работы, выполняемой для прохода по коллекциям в языках более низкого уровня вроде С.

Однако существуют ситуации, когда требуется выполнять проход более специализированными способами. Скажем, что если нужно посетить каждый второй или каждый третий элемент в списке, либо попутно изменять список? Как насчет обхода более одной последовательности параллельно в том же самом цикле `for`? Что если необходима также индексация?

Вы всегда можете записывать такие специфичные итерации с помощью цикла `while` и ручной индексации, но Python предлагает набор встроенных функций, которые позволяют специализировать итерацию в цикле `for`.

- Встроенная функция `range` (доступная, начиная с Python 0.X) производит серию последовательно возрастающих целых чисел, которые могут использоваться в качестве индексов в цикле `for`.
- Встроенная функция `zip` (доступная, начиная с версии Python 2.0) возвращает серию кортежей из параллельных элементов, которые могут применяться для обхода множества последовательностей в цикле `for`.
- Встроенная функция `enumerate` (доступная, начиная с версии Python 2.3) генерирует значения и индексы элементов в итерируемом объекте, так что вести счет вручную не придется.
- Встроенная функция `map` (доступная, начиная с версии Python 1.0) способна давать эффект, похожий на `zip` в Python 2.X, хотя такая ее роль была удалена в Python 3.X.

Тем не менее, поскольку циклы `for` могут выполняться быстрее, чем циклы с подсчетом на основе `while`, в ваших интересах использовать инструменты подобного рода, которые позволяют применять `for`, когда только возможно. Давайте рассмотрим перечисленные встроенные функции по очереди в свете распространенных сценариев использования. Как вы увидите, применение функций может слегка отличаться между Python 2.X и Python 3.X, а некоторые их приложения более обоснованы по сравнению с другими.

Циклы с подсчетом: `range`

Первая связанная с циклами функция, `range`, в действительности является универсальным инструментом, который может использоваться в разнообразных контекстах. Она кратко упоминалась в главе 4. Хотя функция `range` наиболее часто будет применяться для генерации индексов в цикле `for`, вы можете ее использовать где угодно, когда требуется серия целых чисел. В Python 2.X функция `range` создает физический список; в Python 3.X функция `range` является *итерируемым объектом*, который генерирует элементы по запросу, поэтому для отображения сразу всех результатов вызов `range` понадобится поместить внутрь вызова `list`:

```
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

С одним аргументом `range` генерирует список целых чисел, начиная с нуля и заканчивая, но не включая значение аргумента. В случае передачи двух аргументов первый считается нижней границей. Необязательный третий аргумент может задавать `шаг`, когда он указан, Python добавляет выбранный шаг к каждому последующему целому числу в результате (по умолчанию шаг равен +1). При желании диапазоны могут быть неположительными и невозрастающими:

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(range(5, -5, -1))
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Более формально итерируемые объекты вроде такого рассматриваются в главе 14. Там мы также покажем, что в Python 2.X имеется родственная функция по имени `xrange`, которая похожа на `range`, но не создает результирующий список в памяти целиком. Это является оптимизацией пространства, которое включено в Python 3.X генераторным поведением его функции `range`.

Несмотря на то что такие результаты `range` могут быть полезными и сами по себе, они оказываются наиболее удобными внутри циклов `for`. Первым делом они представляют способ для повторения действия указанное количество раз. Например, чтобы вывести три строки, с помощью `range` генерируется соответствующее количество целых чисел:

```
>>> for i in range(3):
...     print(i, 'Pythons')
...
0 Pythons
1 Pythons
2 Pythons
```

Обратите внимание, что циклы `for` в Python 3.X автоматически получают результаты из `range`, а потому вызов `list` здесь применять не нужно (в Python 2.X мы получаем временный список, если только взамен не вызвали `xrange`).

Просмотр последовательностей: `while` и `range` или `for`

Вызов `range` также иногда используется для непрямого прохода по последовательности, хотя часто такой подход не является наилучшим в данной роли. Самый легкий и обычно наиболее быстрый способ прохода по последовательности всегда предусматривает применение простого цикла `for`, т.к. Python самостоятельно поддерживает большинство деталей:

```
>>> X = 'spam'
>>> for item in X: print(item, end=' ')
# Простая итерация
...
s p a m
```

При использовании подобным образом цикл `for` внутренне обрабатывает все детали автоматически. Если вам на самом деле необходимо взять на себя логику индексации, то вы можете сделать это посредством цикла `while`:

```
>>> i = 0
>>> while i < len(X):
...     print(X[i], end=' ')
# Итерация в цикле while
...
i += 1
...
s p a m
```

Однако вы можете делать ручную индексацию также и в цикле `for`, если примените функцию `range` с целью генерации списка индексов для прохода по ним. Процесс многошаговый, но его вполне достаточно для генерации смещений, а не элементов по этим смещениям:

```
>>> X
'spam'
>>> len(X)
4
>>> list(range(len(X)))
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print(X[i], end=' ') # Ручная итерация
# посредством range/len
...
spam
```

Обратите внимание, что поскольку в примере производится проход по списку *смещений* в `X`, а не по действительным элементам `X`, внутри цикла для извлечения каждого элемента необходима индексация в `X`. Тем не менее, если это кажется излишним, то причина в том, что в данном примере настолько трудная работа попросту не нужна.

Хотя комбинации `range/len` в такой роли достаточно, вряд ли стоит считать ее лучшим вариантом. Код может выполняться медленнее и требует большего объема работы, нежели оправдано. Если только не существует специального требования индексации, то лучше использовать простую форму цикла `for` в Python:

```
>>> for item in X: print(item, end=' ')
# По возможности применяйте
# простую итерацию
```

В качестве общего правила запомните: всякий раз, когда это возможно, применяйте `for` вместо `while`, и не используйте вызовы `range` в циклах `for` кроме крайних случаев. Более простое решение практически всегда будет лучшим. Однако, как и в каждом хорошем правиле, есть масса исключений — что и демонстрируется в следующем разделе.

Тасование последовательностей: `range` и `len`

Хотя и не идеально для простого просмотра последовательностей, использованный в предыдущем примере кодовый шаблон позволяет реализовывать специализированные виды обхода, когда они необходимы. Скажем, некоторые алгоритмы могут действовать переупорядочение последовательностей — для генерации вариантов при поиске, для проверки эффекта от разного упорядочения значений и т.д. Такие случаи могут требовать смещения, чтобы разделять последовательности и снова собирать их вместе, как показано ниже; целые числа диапазона предоставляют счетчик повторений в первом и позицию для нарезания во втором:

```
>>> S = 'spam'
>>> for i in range(len(S)):
...     S = S[1:] + S[:1]
...     print(S, end=' ')
...
spams amsp mspa spam

>>> S
'spam'
```

```
>>> for i in range(len(S)):      # Для позиций 0..3
...     X = S[i:] + S[:i]        # Задняя часть + передняя часть
...     print(X, end=' ')
...
spam pams amsp mspa
```

Отследите приведенные циклы по одной итерации за раз, если они кажутся непонятными. Второй цикл создает те же результаты, что и первый, но в другом порядке, не изменяя в ходе дела исходную переменную. Из-за того, что оба производят нарезание, чтобы получить части для объединения, они также работают на последовательности любого типа и возвращают последовательности того же типа, что и тасуемые — когда вы тасуете список, то создаете переупорядоченный список:

```
>>> L = [1, 2, 3]
>>> for i in range(len(L)):
...     X = L[i:] + L[:i]        # Работает на последовательности любого типа
...     print(X, end=' ')
...
[1, 2, 3] [2, 3, 1] [3, 1, 2]
```

Мы будем применять код подобного рода для тестирования функций с разными упорядочениями аргументов в главе 18 и распространим его на функции, генераторы и более искусные перестановки в главе 20 — это крайне полезный инструмент.

Неполный обход: `range` или срезы

Случаи вроде приведенных в предыдущем примере являются эффективными приложениями для комбинации `range/len`. Мы можем также использовать такую методику для пропускания элементов по мере продвижения:

```
>>> S = 'abcdefghijkl'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]
>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
a c e g i k
```

Здесь мы посещаем каждый *второй* элемент в строке `S` за счет перехода согласно списку, сгенерированному функцией `range`. Для посещения каждого третьего элемента измените третий аргумент `range` на 3 и т.д. Фактически применение `range` подобным образом позволяет пропускать элементы в циклах, по-прежнему сохраняя простоту циклической конструкции `for`.

Тем не менее, в наши дни это также вряд ли можно считать рекомендуемой методикой в Python. Если вы действительно намереваетесь пропускать элементы в последовательности, то расширенная форма *выражения среза* с тремя пределами, представленная в главе 7, обеспечивает более простой способ достижения той же цели. Например, для посещения каждого второго символа в `S` нарезайте со страйдом 2:

```
>>> S = 'abcdefghijkl'
>>> for c in S[::2]: print(c, end=' ')
...
a c e g i k
```

Результат будет таким же, но код значительно легче для написания и восприятия другими. Потенциальное преимущество использования `range` связано с занимаемым пространством: срез создает копию строки в Python 2.X и 3.X, тогда как `range` в Python 3.X и `xrange` в Python 2.X не создают список; в случае очень больших строк удается сэкономить память.

Изменение списков: range или включения

Еще одним распространенным местом, где может применяться комбинация `range/len` с оператором `for`, являются циклы, которые изменяют список по мере его обхода. В качестве примера предположим, что необходимо добавить 1 к каждому элементу в списке (возможно, обеспечивая повышение каждому сотруднику в списке из базы данных отдела кадров). Вы можете попробовать решить задачу с помощью простого цикла `for`, но результат, вероятно, окажется не совсем таким, который желательно получить:

```
>>> L = [1, 2, 3, 4, 5]

>>> for x in L:
...     x += 1                         # Изменяет x, но не L
...
>>> L
[1, 2, 3, 4, 5]
>>> x
6
```

Код не вполне работоспособен – он изменяет переменную цикла `x`, но не список `L`. Причина довольно тонкая. При каждом проходе цикла `x` ссылается на очередное целое число, уже извлеченное из списка. Скажем, на первой итерации `x` – целое число 1. На следующей итерации тело цикла устанавливает `x` в другой объект, целое число 2, но не обновляет список, откуда первоначально поступило значение 1; переменная `x` занимает отдельный от списка участок памяти.

Для действительного изменения списка во время прохода по нему нам придется использовать индексы, чтобы можно было присвоить обновленное значение в каждой проходимой позиции. Требуемые индексы способна производить комбинация `range/len`:

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):
...     L[i] += 1                      # Добавить 1 к каждому элементу в L
...                               # Или L[i] = L[i] + 1
...
>>> L
[2, 3, 4, 5, 6]
```

При таком подходе список изменяется по мере выполнения цикла. Нет никакой возможности сделать то же самое посредством простого цикла в стиле `for x in L:`, потому что такой цикл проходит по фактическим элементам, а не по позициям списка. Но как насчет эквивалентного цикла `while`? Он потребует чуть большей работы с нашей стороны и может выполнять медленнее в зависимости от версии (медленнее в Python 2.7 и 3.3, хотя не до такой степени в Python 3.3 – мы увидим, как это проверить, в главе 21):

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

Однако и здесь решение с `range` также может оказаться не самым идеальным. Списковое включение вида:

```
[x + 1 for x in L]
```

в наши дни, вероятно, выполняется быстрее и делает похожую работу, хотя и без изменения исходного списка на месте (мы могли бы присвоить новый списковый объект выражения обратно `L`, но это не обновило бы любые другие ссылки на исходный список). Списковые включения будут детально исследоваться в следующей главе.

Параллельные обходы: `zip` и `map`

Рассматриваемая в настоящем разделе методика расширяет область действия цикла. Как вы видели,строенная функция `range` позволяет обходить последовательности с помощью цикла `for` в неполной манере. В том же духестроенная функция `zip` дает возможность применять циклы `for` для просмотра множества последовательностей *параллельно* – без совмещения во времени, но в течение того же самого цикла. В базовом виде функция `zip` принимает одну или большее количество последовательностей в качестве аргументов и возвращает серию кортежей, объединяющих в пары параллельные элементы из указанных последовательностей. Предположим, что мы работаем с двумя списками (возможно, списком имен и списком адресов, согласующихся по позициям):

```
>>> L1 = [1, 2, 3, 4]
>>> L2 = [5, 6, 7, 8]
```

Чтобы объединить элементы в таких списках, мы можем использовать `zip` для создания списка кортежей с парами. Подобно `range` функция `zip` является списком в Python 2.X, но итерируемым объектом в Python 3.X, где ее придется помещать внутрь вызова `list` для отображения сразу всех результатов (итерируемые объекты более подробно обсуждаются в следующей главе):

```
>>> zip(L1, L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2))      # list() требуется в Python 3.X, но не в Python 2.X
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Такой результат может быть полезен и в других контекстах, но в сочетании с циклом `for` он поддерживает параллельные итерации:

```
>>> for (x, y) in zip(L1, L2):
...     print(x, y, '--', x+y)
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

Здесь мы проходим по результату вызова `zip`, т.е. по парам элементов, извлеченных из двух списков. Обратите внимание, что данный цикл `for` снова применяет представленную ранее форму присваивания кортежей для распаковки каждого кортежа в результате `zip`. При первом проходе получается так, как если бы мы выполняли оператор присваивания `(x, y) = (1, 5)`.

Совокупный эффект заключается в том, что мы просматриваем в цикле оба списка, `L1` и `L2`. Похожего результата можно было бы добиться с помощью цикла `while`, который поддерживает индексацию вручную, но он потребовал бы большего объема набора и почти наверняка выполнялся бы медленнее, чем подход с `for`/`zip`.

Строго говоря, функция `zip` более универсальна, чем вытекает из приведенного примера. Скажем, она принимает последовательность любого типа (на самом деле любой итерируемый объект, включая файлы) и допускает передачу более двух аргументов. В случае трех аргументов, как в следующем примере, она строит список трехэлементных кортежей с элементами из каждой последовательности, по существу проецируя по столбцам (формально для N аргументов мы получаем N -арный кортеж):

```
>>> T1, T2, T3 = (1, 2, 3), (4, 5, 6), (7, 8, 9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3))           # Три кортежа для трех аргументов
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Кроме того, `zip` усекает результирующие кортежи по длине самой короткой последовательности, когда длины аргументов отличаются. В показанном ниже коде мы используем функцию `zip` для двух строк, чтобы выбирать символы параллельно, но результат будет содержать столько кортежей, сколько символов в более короткой строке:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2))           # Усекает по длине более короткой строки
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Эквивалентность `map` в Python 2.X

В Python 2.X имеется связанный встроенный функция `map`, которая объединяет в пары элементы из последовательностей в похожей манере при передаче ей `None` в первом аргументе (для функции), но она дополняет более короткие последовательности с помощью `None`, если длины аргументов отличаются, а не усекает до наименьшей длины:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> map(None, S1, S2)           # Python 2.X: дополняет до наибольшей длины
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

В примере применяется вырожденная форма встроенной функции `map`, которая больше не поддерживается в Python 3.X. Обычно `map` принимает функцию и один или более аргументов последовательностей и накапливает результаты, вызывая функцию с параллельными элементами, которые извлекаются из последовательности (последовательностей).

Функция `map` будет детально исследоваться в главах 19 и 20, но в качестве короткого примера в показанном ниже коде `map` применяет встроенную функцию `ord` ко всем элементам в строке и накапливает результаты (подобно `zip`, функция `map` является генератором значений в Python 3.X и потому должна быть помещена в вызов `list` для отображения сразу всех результатов):

```
>>> list(map(ord, 'spam'))
[115, 112, 97, 109]
```

Код работает точно так же, как следующий оператор цикла, но `map` часто выполняется быстрее, как вы увидите в главе 21:

```
>>> res = []
>>> for c in 'spam': res.append(ord(c))
>>> res
[115, 112, 97, 109]
```



Примечание, касающееся нестыковки версий. Вырожденная форма map, использующая None для аргумента функции, в Python 3.X больше не поддерживается, поскольку она в значительной степени совпадает с zip (и откровенно говоря, несколько не в ладах с прикладным назначением самой функции map). В Python 3.X либо применяйте zip, либо пишите код цикла, чтобы самостоятельно дополнять результаты. Фактически мы посмотрим, как писать такой код цикла в главе 20 после того, как проясним ряд дополнительных концепций, связанных с итерацией.

Создание словарей с помощью zip

Давайте взглянем на еще один сценарий использования zip. В главе 8 было высказана мысль, что применяемый здесь вызов zip также может оказаться удобным для генерации словарей, когда наборы ключей и значений должны вычисляться во время выполнения. Теперь, когда мы освоились с функцией zip, имеет смысл посмотреть, каким образом она соотносится с созданием словарей. Как вам известно, словарь всегда можно создавать, записывая словарный литерал или присваивая по ключам:

```
>>> D1 = {'spam':1, 'eggs':3, 'toast':5}
>>> D1
{'eggs': 3, 'toast': 5, 'spam': 1}

>>> D1 = {}
>>> D1['spam'] = 1
>>> D1['eggs'] = 3
>>> D1['toast'] = 5
```

Тем не менее, что делать, если программа получает ключи и значения словаря в списках во время выполнения, т.е. после того, как сценарий написан? Скажем, пусть имеются следующие списки ключей и значений, собранные от пользователя, полученные в результате разбора содержимого файла либо извлеченные из другого динамического источника:

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]
```

Одно из решений по превращению таких списков в словарь предусматривало бы вызов zip для списков и проход по ним параллельно посредством цикла for:

```
>>> list(zip(keys, vals))
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'eggs': 3, 'toast': 5, 'spam': 1}
```

Однако оказывается, что в Python 2.2 и последующих версиях можно вообще опустить цикл for и просто передать объединенные с помощью zip списки ключей/значений вызову встроенного конструктора dict:

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]

>>> D3 = dict(zip(keys, vals))
>>> D3
{'eggs': 3, 'toast': 5, 'spam': 1}
```

Встроенное имя `dict` на самом деле является именем типа в Python (более подробные сведения об именах типов и создании подклассов вы найдете в главе 32). Вызов `dict` приводит к чему-то вроде преобразования списков в словарь, но в действительности он представляет собой запрос конструирования объекта.

В следующей главе мы исследуем связанную, но более широкую концепцию спискового включения, которое строит списки в единственном выражении; мы снова обратимся к включениям словарей Python 3.X и 2.7 – альтернативе вызову `dict` для объединенных посредством `zip` пар ключей и значений:

```
>>> {k: v for (k, v) in zip(keys, vals)}
{'eggs': 3, 'toast': 5, 'spam': 1}
```

Генерация смещений и элементов: `enumerate`

Последняя рассматриваемая вспомогательная функция предназначена для поддержки двойного режима использования. Ранее мы обсуждали применение `range` для генерации смещений элементов в строке, а не самих элементов по этим смещениям. Тем не менее, в ряде случаев нам необходимо то и другое: элемент для использования и смещение по мере продвижения. Традиционно такая задача решалась посредством простого цикла `for`, который также вел счетчик текущего смещения:

```
>>> S = 'spam'
>>> offset = 0
>>> for item in S:
...     print(item, 'appears at offset', offset) # Вывод элемента и смещения
...     offset += 1
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

Прием работает, но в недавних выпусках Python 2.X и 3.X (начиная с версии Python 2.3) подобную работу делает новая встроенная функция по имени `enumerate` – она снабжает циклы счетчиком “бесплатно”, не заставляя приносить в жертву простоту автоматической итерации:

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S):
...     print(item, 'appears at offset', offset) # Вывод элемента и смещения
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

Функция `enumerate` возвращает *генераторный объект* – разновидность объекта, поддерживающий протокол итерации, который мы исследуем в следующей главе и обсудим более подробно в части IV книги. Такой объект имеет метод, вызываемый встроенной функцией `next`, который на каждом проходе цикла возвращает кортеж (*индекс, значение*). Цикл `for` проходит по этим кортежам автоматически, что позволяет распаковывать их значения с помощью присваивания кортежей почти так, как мы делали для `zip`:

```
>>> E = enumerate(S)
>>> E
```

```
<enumerate object at 0x0000000002A8B900>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

Обычно мы не видим данный механизм, т.к. все контексты итерации, в том числе списковые включения, рассматриваемые в главе 14, автоматически следуют протоколу итерации:

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']

>>> for (i, l) in enumerate(open('test.txt')):
...     print('%s) %s' % (i, l.rstrip()))
...
0) aaaaaa
1) bbbbbbb
2) cccccc
```

Однако для полного понимания концепций итерации вроде `enumerate`, `zip` и списковых включений необходим более формальный анализ, который проводится в следующей главе.

Что потребует внимания: команды оболочки и другое

В предыдущей врезке были показаны циклы, примененные к файлам. Как кратко отмечалось в главе 9, связанный вызов `os.popen` в Python также предоставляет интерфейс, подобный файлам, для чтения вывода, порождаемого *командами оболочки*. Поскольку вы уже освоили операторы цикла, ниже приведен пример упомянутого инструмента в действии — чтобы запустить команду оболочки и прочитать текст из ее стандартного вывода, передайте команду в виде строки вызову `os.popen` и выполните чтение из возвращенного им объекта, подобного файлу (в случае возникновения проблемы с кодировкой Unicode решить ее может помочь обсуждение в главе 25):

```
>>> import os
>>> F = os.popen('dir')                      # Чтение строки за строкой
>>> F.readline()
' Volume in drive C has no label.\n'
>>> F = os.popen('dir')                      # Чтение блоками заданного размера
>>> F.read(50)
' Volume in drive C has no label.\n Volume Serial Nu'

>>> os.popen('dir').readlines()[0]            # Чтение всех строк: индексация
' Volume in drive C has no label.\n'
>>> os.popen('dir').read()[:50]                # Чтение всего сразу: срез
' Volume in drive C has no label.\n Volume Serial Nu'

>>> for line in os.popen('dir'):              # Цикл с файловым итератором строк
...     print(line.rstrip())
...
Volume in drive C has no label.
Volume Serial Number is D093-D1F7
...и так далее...
```

В коде запускается команда вывода списка файлов каталога (`dir`) в Windows, но подобным образом можно запустить любую программу, которая допускает запуск из командной строки. Мы могли бы использовать такую схему, например, для отображения вывода команды `systeminfo` в Windows – `os.system` просто запускает команду оболочки, но `os.popen` также подключается к ее потокам; оба показанных далее фрагмента показывают вывод команды оболочки в простом консольном окне, но первый может не отобразиться в графическом пользовательском интерфейсе, таком как IDLE:

```
>>> os.system('systeminfo')
...вывод в консоли, всплывающее окно в IDLE...
0
>>> for line in os.popen('systeminfo'): print(line.rstrip())

Host Name: MARK-VAIO
OS Name: Microsoft Windows 7 Professional
OS Version: 6.1.7601 Service Pack 1 Build 7601
...дополнительный текст со сведениями о системе...
```

К полученному выводу команды в текстовой форме применим любой инструмент или методика обработки строк, включая форматирование для отображения и разбор содержимого:

```
# Форматирование, ограниченное отображение
>>> for (i, line) in enumerate(os.popen('systeminfo')):
...     if i == 4: break
...     print('%05d) %s' % (i, line.rstrip()))
...
00000)
00001) Host Name:      MARK-VAIO
00002) OS Name:        Microsoft Windows 7 Professional
00003) OS Version:     6.1.7601 Service Pack 1 Build 7601

# Разбор для специфических строк, нейтральный к регистру символов
>>> for line in os.popen('systeminfo'):
...     parts = line.split(':')
...     if parts and parts[0].lower() == 'system type':
...         print(parts[1].strip())
...
x64-based PC
```

Мы увидим метод `os.popen` в действии в главе 21, когда прибегнем к нему при чтении результатов сконструированной командной строки, которая фиксирует продолжительность выполнения вариантов кода, и в главе 25, где он будет использоваться при сравнении выводов тестируемых сценариев.

Инструменты наподобие `os.popen` и `os.system` (а также не показанный здесь модуль `subprocess`) позволяют задействовать любую программу командной строки на компьютере, но можно также писать эмуляторы с внутрипроцессным кодом. Скажем, эмуляция способности Unix-утилиты `awk` по извлечению столбцов из текстовых файлов в Python почти тривиальна и может стать многократно применяемой функцией в процессе:

```
# Эмуляция awk: извлечение столбца 7 из файла с разделителями
# в виде пробельных символов
for val in [line.split()[6] for line in open('input.txt')]:
    print(val)
```

```

# То же самое, но более явный код, который предохраняет результат
col7 = []
for line in open('input.txt'):
    cols = line.split()
    col7.append(cols[6])
for item in col7: print(item)

# То же самое, но многократно используемая функция (см. следующую часть книги)
def awker(file, col):
    return [line.rstrip().split()[col-1] for line in open(file)]

print(awker('input.txt', 7))          # Список строк
print('.join(awker('input.txt', 7)))  # Поместить запятые между строками

```

Тем не менее, Python сам по себе предоставляет доступ, подобный файлу, к широкому многообразию данных, в том числе к тексту, возвращаемому *веб-сайтами* и их страницами по URL, хотя рассмотрение деталей используемого здесь импорта пакетов и других ресурсов об инструментах в целом мы отложим до части V (например, следующий код работает в Python 2.X, но применяет `urllib` вместо `urllib.request` и возвращает текстовые строки):

```

>>> from urllib.request import urlopen
>>> for line in urlopen('http://learning-python.com/books'):
...     print(line)
...
b'<HTML>\n'
b'\n'
b'<HEAD>\n'
b"<TITLE>Mark Lutz's Book Support Site</TITLE>\n"
...etc...

```

Резюме

В главе мы исследовали операторы цикла Python, а также ряд концепций, имеющих отношение к организации циклов в Python. Мы подробно рассмотрели операторы `while` и `for`, а также прояснили назначение связанных с ними конструкций `else`. Кроме того, мы изучили операторы `break` и `continue`, которые имеют смысл только внутри циклов, и ознакомились с несколькими встроенными инструментами, часто используемыми в циклах `for`, в том числе `range`, `zip`, `map` и `enumerate`, хотя определенные детали, касающиеся их ролей как итерируемых объектов в Python 3.X, были намеренно сокращены.

В следующей главе мы продолжим историю об итерации обсуждением списковых включений и протокола итерации — концепций, прочно связанных с циклами `for`. Там мы также обрисуем остаток картины, лежащей в основе встреченных здесь итерируемых инструментов, таких как `range` и `zip`, и ознакомимся с тонкостями их работы. Как всегда, прежде чем двигаться дальше, закрепите пройденный материал, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Каковы главные функциональные отличия между `while` и `for`?
2. В чем разница между `break` и `continue`?
3. Когда выполняется конструкция `else` цикла?
4. Как можно записать цикл с подсчетом в Python?
5. Для чего может применяться `range` в цикле `for`?

Проверьте свои знания: ответы

1. `while` является универсальным оператором цикла, но `for` предназначен для итерации по элементам в последовательности или другом итерируемом объекте. Хотя цикл `while` может имитировать циклы `for` с подсчетом, он требует большего объема кода и может выполняться медленнее.
2. Оператор `break` производит немедленный выход из цикла (в точку непосредственно после оператора цикла `while` или `for`), а оператор `continue` инициирует переход в начало цикла (прямо перед проверкой в `while` либо извлечением следующего элемента в `for`).
3. Конструкция `else` в цикле `while` или `for` будет выполняться один раз при выходе из цикла, если выход произошел нормально (без выполнения оператора `break`). Оператор `break` обеспечивает незамедлительный выход из цикла, пропуская часть `else` (при ее наличии).
4. Циклы с подсчетом можно записывать с помощью оператора `while`, который поддерживает ручную индексацию, или оператора `for`, использующего встроенную функцию `range` для генерации последовательных целочисленных смещений. Ни один из них не является предпочтительным способом работы в Python, если вам необходимо просто пройти по элементам в последовательности. Взамен когда только возможно применяйте простой цикл `for` без `range` или подсчета; его легче записывать и обычно он быстрее выполняется.
5. Встроенная функция `range` может использоваться в цикле `for` для реализации фиксированного количества повторений, для просмотра по смещениям вместо элементов по этим смещениям, для пропуска следующих друг за другом элементов в ходе дела и для изменения списка во время прохода по нему. Ни одна из этих ролей не требует `range`, а большинство имеют альтернативы — просмотр фактических элементов, срезы с тремя границами и списковые включения в наши дни часто будут лучшими решениями (вопреки природной склонности бывших программистов на С к подсчету всего, что угодно!).

Итерации и включения

В предыдущей главе вы ознакомились с двумя операторами цикла Python — `while` и `for`. Хотя они могут справиться с большинством повторяющихся задач, которые необходимо выполнять в программах, потребность в итерации по последовательностям оказывается настолько распространенной и всепроникающей, что Python предлагает дополнительные инструменты, чтобы сделать итерацию проще и эффективнее. В этой главе мы начинаем исследование таких инструментов. В частности, мы представим концепции *протокола итерации* Python — модели на основе вызова методов, используемой циклом `for`, и восполним недостающие детали о *списковых включениях*, которые являются близким родственником цикла `for`, применяющего выражение к элементам в итерируемом объекте.

Поскольку указанные инструменты относятся как к циклу `for`, так и к функциям, мы примем для их раскрытия в книге двухпроходный подход вместе с заключением.

- Текущая глава знакомит с их основами в контексте инструментов циклов, выступая в качестве своего рода продолжения предыдущей главы.
- В главе 20 они пересматриваются в контексте инструментов, базирующихся на функциях, и тематика расширяется рассмотрением встроенных функций и определяемых пользователем генераторов.
- В главе 30 (том 2) предлагается финальная часть истории, посвященная итерируемым объектам, определяемым пользователем, которые записываются с помощью классов.

В настоящей главе мы также испытаем дополнительные итерационные инструменты в Python и коснемся темы новых итерируемых объектов, доступных в Python 3.X, где понятие итерируемых объектов получило еще большее распространение.

Одно заблаговременное замечание: некоторые концепции, представленные в упомянутых выше главах, на первых взглядах могут показаться слишком продвинутыми. Однако по мере приобретения опыта вы обнаружите, что эти инструменты полезны и эффективны. Хотя они не являются строго обязательными, из-за частого использования таких инструментов в коде Python знание их основ поможет лучше понимать код, написанный другими.

Итерации: первый взгляд

В предыдущей главе было указано, что цикл `for` может работать с любым типом последовательности в Python, включая списки, кортежи и строки, например:

```
>>> for x in [1, 2, 3, 4]: print(x ** 2, end=' ') # В Python 2.X: print x ** 2,  
***  
1 4 9 16  
>>> for x in (1, 2, 3, 4): print(x ** 3, end=' ')  
***  
1 8 27 64  
>>> for x in 'spam': print(x * 2, end=' ')  
***  
ss pp aa mm
```

На самом деле цикл `for` способен быть даже более универсальным — он работает на любом *итерируемом объекте*. Фактически сказанное справедливо для всех итерационных инструментов в Python, которые просматривают объекты слева направо, в том числе циклов `for`, рассматриваемых в этой главе списковых включений, проверок членства `in`, встроенной функции `map` и т.д.

Концепция “итерируемых объектов” появилась в Python относительно недавно, но она буквально пронизывает всю языковую модель. По существу она представляет собой обобщение понятия последовательностей — объект считается *итерируемым*, если он является или физически хранящейся последовательностью, или объектом, который производит по одному результату за раз в контексте итерационного инструмента, подобного циклу `for`. В определенном смысле итерируемые объекты включают и физические последовательности, и *виртуальные последовательности*, вычисляемые по требованию.



Терминология в данной теме отличается некоторой свободой. При ссылке на объект, который вообще поддерживает итерацию, термины “итерируемый объект” и “итератор” временами используются взаимозаменяюще. Ради ясности в книге отдаётся предпочтение применению термина *итерируемый объект* в отношении объекта, поддерживающего вызов `iter`, и термина *итератор* для ссылки на объект, который возвращается вызовом `iter` на итерируемом объекте и поддерживает вызов `next(I)`. Оба вызова определяются далее.

Тем не менее, принятое соглашение не является универсальным ни в мире Python, ни в настоящей книге; “итератор” также используется для обозначения инструментов, которые выполняют итерацию. В главе 20 данная категория расширяется термином “генератор”, который относится к объектам, автоматически поддерживающим протокол итерации, и потому итерируемым — даже притом, что все итерируемые объекты генерируют результаты!

Протокол итерации: итераторы файловых объектов

Протокол итерации легче всего понять, посмотрев, как он работает со встроенным типом, таким как файл. Для демонстрационных целей в этой главе мы будем применять следующий входной файл:

```
>>> print(open('script2.py').read())  
import sys  
print(sys.path)  
x = 2  
print(x ** 32)  
  
>>> open('script2.py').read()  
'import sys\nprint(sys.path)\n\nx = 2\nprint(x ** 32)\n'
```

Вспомните из главы 9, что открытые файловые объекты имеют метод по имени `readline`, который читает из файла одну строку текста за раз – при каждом вызове метода `readline` мы переходим на следующую строку. При достижении конца файла возвращается пустая строка, которую можно выявлять для выхода из цикла:

```
>>> f = open('script2.py')      # Чтение четырехстрочного файла сценария
                                         # в текущем каталоге
>>> f.readline()      # При каждом вызове метод readline загружает одну строку
'import sys\n'
>>> f.readline()
'print(sys.path)\n'
>>> f.readline()
'x = 2\n'
>>> f.readline()      # Последние строки могут иметь или не иметь \n
'print(x ** 32)\n'
>>> f.readline()      # При достижении конца файла возвращается пустая строка
''
```

Однако файлы также располагают методом по имени `__next__` в Python 3.X (и `next` в Python 2.X), который имеет почти идентичный эффект – каждый раз, когда `__next__` вызывается, он возвращает следующую строку из файла. Единственное заметное отличие в том, что при достижении конца файла метод `__next__` генерирует встроенное исключение `StopIteration`, а не возвращает пустую строку:

```
>>> f = open('script2.py')      # При каждом вызове метод __next__
                                         # также загружает одну строку
>>> f.__next__()      # Но при достижении конца файла генерирует исключение
'import sys\n'
>>> f.__next__()      # Используйте f.next() в Python 2.X
                                         # либо next(f) в Python 2.X или 3.X
'print(sys.path)\n'
>>> f.__next__()
'x = 2\n'
>>> f.__next__()
'print(x ** 32)\n'
>>> f.__next__()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Данный интерфейс является большей частью того, что мы называем *протоколом итерации* в Python. Любой объект с методом `__next__` для перехода на следующий результат, который генерирует исключение `StopIteration` при достижении конца серии результатов, в Python считается итератором. По любому такому объекту можно также выполнять проход с помощью цикла `for` или другого итерационного инструмента, потому что все внутренние итерационные инструменты обычно работают, вызывая `__next__` на каждой итерации и перехватывая исключение `StopIteration` для выяснения, когда выходить. Как вскоре будет показано, для некоторых объектов полный протокол включает дополнительный первый шаг, связанный с вызовом `iter`, но для файлов он не требуется.

Совокупный эффект описанной магии состоит в том, что согласно утверждению, сделанному в главах 9 и 13, наилучший на сегодняшний день способ чтения текстового файла строка за строкой – *не читать его вообще*, а взамен позволить циклу `for` автоматически вызывать метод `__next__` для перехода к следующей строке на каждой итерации. В ходе дела итератор файлового объекта будет выполнять работу по автоматической

загрузке строк. Скажем следующий код читает файл строка за строкой, попутно выводя версию каждой строки в верхнем регистре, причем вообще без явного чтения из файла:

```
>>> for line in open('script2.py'):
...     print(line.upper(), end='')
... 
... IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

Обратите внимание, что `end=''` в `print` здесь используется для того, чтобы подавить добавление символа `\n`, т.к. строки уже его содержат (без `end=''` строки вывода будут чередоваться с пустыми строками; хвостовая запятая в Python 2.X делает ту же работу, что и `end`). В наши дни такой способ считается *наилучшим* для чтения текстовых файлов строка за строкой по трем причинам: он самый простой в плане написания кода, может оказаться самым быстрым при выполнении и лучший в отношении затрат памяти. Более старый, первоначальный способ достижения того же эффекта посредством цикла `for` предусматривает вызов файлового метода `readlines` для загрузки содержимого файла в память в виде списка строк:

```
>>> for line in open('script2.py').readlines():
...     print(line.upper(), end='')
... 
... IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

Прием с методом `readlines` по-прежнему работает, но теперь не рассматривается как рекомендуемый подход и хуже в том, что касается расходования памяти. В действительности, поскольку эта версия фактически загружает в память сразу весь файл, он даже не будет работать для файлов, которые слишком велики, чтобы уместиться в пространстве памяти, доступном на компьютере. Напротив, из-за чтения по одной строке за раз версия на основе итератора неуязвима для такого рода проблем бурного роста расходуемой памяти. Кроме того, версия с итератором способна выполнять быстрее, хотя скорость может варьироваться от выпуска к выпуску.

Как упоминалось во врезке “Что потребует внимания: приемы просмотра файлов” в главе 13, читать файл строка за строкой возможно также с помощью цикла `while`:

```
>>> f = open('script2.py')
>>> while True:
...     line = f.readline()
...     if not line: break
...     print(line.upper(), end='')
...
... тот же самый вывод...
```

Тем не менее, такой код может выполняться медленнее, чем версия на основе итератора с циклом `for`, потому что итераторы работают внутри Python со скоростью скомпилированного кода C, а версия с циклом `while` запускает байт-код Python через виртуальную машину Python. Каждый раз, когда мы меняем код Python на код C, скорость имеет тенденцию возрастать. Однако это не абсолютная истина, особенно

в Python 3.X; позже в главе 21 мы увидим методики измерения времени для определения относительной скорости альтернативных версий вроде показанных выше¹.



Примечание, касающееся нестыковки версий. В Python 2.X метод итерации называется `X.next()`, а не `X.__next__()`. Для совместимости встроенная функция `next(X)` также доступна в Python 3.X и 2.X (в Python 2.6 и последующих версиях); она вызывает `X.__next__()` в Python 3.X и `X.next()` в Python 2.X. Помимо имен методов во всех других отношениях итерация в Python 2.X и 3.X работает одинаково. В версиях Python 2.6 и 2.7 для ручных итераций просто применяйте `X.next()` или `next(X)` вместо `X.__next__()` из Python 3.X; в версиях, предшествующих Python 2.6, используйте вызовы `X.next()` вместо `next(X)`.

Ручная итерация: `iter` И `next`

Чтобы упростить написание кода ручной итерации, Python 3.X также предлагает встроенную функцию `next`, которая автоматически вызывает метод `__next__` объекта. Согласно предыдущей врезке “На заметку!” ради совместимости этот вызов также поддерживается в Python 2.X. Для заданного объекта итератора `X` вызов `next(X)` является тем же самым, что и `X.__next__()` в Python 3.X (и `X.next()` в Python 2.X), но он значительно проще и более нейтрален к версиям. Например, с файлами можно применять любую из двух форм:

```
>>> f = open('script2.py')
>>> f.__next__()      # Вызов итерационного метода напрямую
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'

>>> f = open('script2.py')
>>> next(f)        # В Python 3.X встроенная функция next(f) вызывает f.__next__()
'import sys\n'
>>> next(f)        # next(f) => [Python 3.X: f.__next__()], [Python 2.X: f.next()]
'print(sys.path)\n'
```

Формально есть еще один аспект протокола итерации, который упоминался ранее. Когда цикл `for` начинается, он сначала получает итератор от итерируемого объекта, передавая его встроенной функции `iter`; возвращаемый функцией `iter` объект в свою очередь имеет обязательный метод `next`. Функция `iter` внутренне вызывает метод `__iter__` во многом подобно `next` и `__next__`.

Полный протокол итерации

В качестве более формального определения на рис. 14.1 приведена упрощенная структура полного протокола итерации, который используется каждым итерационным инструментом в Python и поддерживается широким разнообразием типов объектов.

¹ Интрига отменяется: итератор файлового объекта по-прежнему кажется чуть быстрее, чем метод `readlines`, и минимум на 30% быстрее цикла `while` в Python 2.7 и 3.3, как показывают тесты, которые я прогонял для кода данной главы на файле с 1000 строк (в Python 2.7 `while` в два раза медленнее). Применимы обычные оговорки при оценочных испытаниях – это верно только для моих версий Python, моего компьютера и моего тестового файла, а Python 3.X усложняет такой анализ за счет переписывания библиотек ввода-вывода с целью поддержки текста Unicode и уменьшения зависимости от системы. В главе 21 раскрываются инструменты и методики, которые вы можете использовать для самостоятельного измерения времени указанных операторов цикла.

В действительности он основан на двух объектах, применяемых итерационными инструментами на двух отдельных шагах:

- итерируемый объект, для которого запрашивается итерация, чей метод `__iter__` запускается методом `iter`;
- объект итератора, возвращенный итерируемым объектом, который фактически производит значения во время итерации, чей метод `__next__` запускается методом `next`, и генерирует исключение `StopIteration`, когда завершает выдачу результатов.

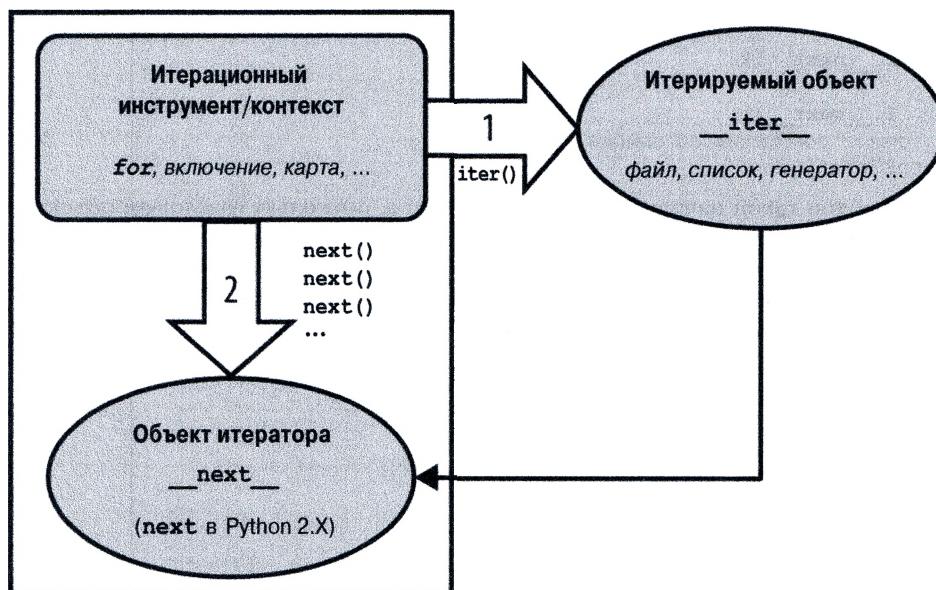


Рис. 14.1. Протокол итерации Python, используемый циклами `for`, включениями, картами и т.д., который поддерживается файлами, списками, словарями, генераторами из главы 20 и другими объектами. Некоторые объекты представляют собой одновременно итерационный контекст и итерируемый объект, такие как генераторные выражения и варианты ряда инструментов в Python 3.X (вроде `map` и `zip`). Определенные объекты являются и итерируемыми, и итераторами, возвращая самих себя для вызова `iter()`, который затем будет пустой операцией

В большинстве случаев такие шаги организуются итерационными инструментами автоматически, но полезно понимать роли упомянутых двух объектов. Скажем, в ряде ситуаций эти два объекта представляют собой одно и то же, когда поддерживается только одиночный просмотр (например, файлы), а объект **итератора** часто является временным, используемым внутренне итерационным инструментом.

Более того, определенные объекты оказываются как инструментом контекста итерации (они выполняют итерацию), так и итерируемым объектом (их результаты итерируются), включая генераторные выражения из главы 20, а также `map` и `zip` в Python 3.X. Как мы увидим позже, в Python 3.X состав итерируемых объектов пополнился дополнительными инструментами, в том числе `map`, `zip`, `range` и некоторыми

словарными методами – чтобы избежать создания полных результирующих списков целиком в памяти.

В реальном коде первый шаг протокола становится очевидным, если взглянуть на то, как циклы `for` внутренне обрабатывают встроенные типы последовательностей, такие как списки:

```
>>> L = [1, 2, 3]
>>> I = iter(L)          # Получение объекта итератора из итерируемого объекта
>>> I.__next__()        # Вызов метода next итератора для продвижения
                           #   на следующий элемент
1
>>> I.__next__()        # Или применение I.next() в Python 2.X, next(I) в любой линейке
2
>>> I.__next__()
3
>>> I.__next__()
...текст сообщения об ошибке не показан...
StopIteration
```

Для файлов такой начальный шаг не требуется, поскольку файловый объект является итератором сам по себе. Из-за того, что файлы поддерживают только одну итерацию (они не допускают перемещение в обратном направлении, чтобы поддерживать множество активных просмотров), файловый объект имеет собственный метод `__next__` и не нуждается в возвращении особого объекта, который предоставлял бы этот метод:

```
>>> f = open('script2.py')
>>> iter(f) is f
True
>>> iter(f) is f.__iter__()
True
>>> f.__next__()
'import sys\n'
```

Тем не менее, списки и многие другие встроенные объекты не являются итераторами сами по себе, потому что они поддерживают множество открытых итераций – например, они могут участвовать в нескольких итерациях во вложенных циклах, причем находится в совершенно разных позициях. Для таких объектов мы обязаны вызывать метод `iter` для запуска итерации:

```
>>> L = [1, 2, 3]
>>> iter(L) is L
False
>>> L.__next__()
AttributeError: 'list' object has no attribute '__next__'
Ошибка атрибута: объект list не имеет атрибута __next__
>>> I = iter(L)
>>> I.__next__()
1
>>> next(I)    # То же, что и I.__next__()
2
```

Ручная итерация

Хотя итерационные инструменты Python вызывают такие функции автоматически, мы можем также использовать их для применения протокола итерации *вручную*.

В следующем взаимодействии демонстрируется эквивалентность автоматической и ручной итераций²:

```
>>> L = [1, 2, 3]
>>>
>>> for X in L:                      # Автоматическая итерация
...     print(X ** 2, end=' ')          # Получает iter, вызывает __next__,
...                                # перехватывает исключения
...
1 4 9
>>> I = iter(L)                     # Ручная итерация: то, что обычно делают циклы for
>>> while True:
...     try:                          # Оператор try перехватывает исключения
...         X = next(I)              # Или вызов I.__next__ в Python 3.X
...     except StopIteration:
...         break
...     print(X ** 2, end=' ')
...
1 4 9
```

Чтобы понять приведенный код, необходимо знать, что операторы *try* запускают действие и перехватывают исключения, которые возникают во время выполнения действия (мы кратко упоминали об исключениях в главе 11, но детально исследуем их в части VII). Я должен также отметить, что циклы *for* и другие контексты итерации иногда могут работать по-разному для классов, определяемых пользователем, многократно индексируя объект вместо следования протоколу итерации, но отдают предпочтение протоколу итерации, если он используется. Более подробные сведения будут даны в главе 30 после ознакомления с перегрузкой операций в классах.

Итерируемые объекты других встроенных типов

Вдобавок к файлам и физическим последовательностям наподобие списков другие типы также располагают полезными итераторами. Скажем, классический способ прохода по ключам в *словаре* предусматривает явное запрашивание его ключей:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> for key in D.keys():
...     print(key, D[key])
...
a 1
b 2
c 3
```

Однако в последних версиях Python словари являются итерируемыми объектами с итератором, который автоматически возвращает по одному ключу за раз в контексте итерации:

```
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'b'
```

² Говоря формально, цикл *for* вызывает внутренний эквивалент *I.__next__*, а не используемый здесь метод *next(I)*, но между ними редко есть какие-то отличия. При ручной итерации, как правило, можно применять любую из двух схем вызова.

```
>>> next(I)
'c'
>>> next(I)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Трассировка (самый последний вызов указан последним) :
  Файл <stdin>, строка 1, в <модуль>
StopIteration
```

Совокупный эффект состоит в том, что нам больше не нужно вызывать метод `keys` для прохода по ключам словаря — цикл `for` будет применять протокол итерации для захвата одного ключа на каждой итерации:

```
>>> for key in D:
...     print(key, D[key])
...
a 1
b 2
c 3
```

Мы не можем здесь углубляться в мельчайшие детали, но отметим, что объекты других типов Python также поддерживают протокол итерации и потому могут использоваться в циклах `for`. Например, хранилища, реализуемые посредством модуля `shelve` (файловая система с доступом по ключу для объектов Python), и результаты, получаемые из `os.popen` (инструмент для чтения вывода команд оболочки, который обсуждался в предыдущей главе), также представляют собой итерируемые объекты:

```
>>> import os
>>> P = os.popen('dir')
>>> P.__next__()
' Volume in drive C has no label.\n'
>>> P.__next__()
' Volume Serial Number is D093-D1F7\n'
>>> next(P)
TypeError: _wrap_close object is not an iterator
Ошибка типа: объект _wrap_close не является итератором
```

Обратите внимание, что в Python 2.X объекты `popen` сами поддерживают метод `P.next()`. В Python 3.X они поддерживают метод `P.__next__()`, но не встроенную функцию `next(P)`. Поскольку функция `next(P)` определена для вызова метода `P.__next__()`, это может казаться необычным, хотя оба вызова работают корректно, если задействован полный протокол итерации, который автоматически применяется циклами `for` и другими итерационными контекстами, с его вызовом `iter` верхнего уровня (что выполняет внутренние шаги, требующиеся из-за поддержки также и вызовов `next` для данного объекта):

```
>>> P = os.popen('dir')
>>> I = iter(P)
>>> next(I)
' Volume in drive C has no label.\n'
>>> I.__next__()
' Volume Serial Number is D093-D1F7\n'
```

Кроме того, в области, связанной с системами, стандартный инструмент прохода по каталогам в Python, `os.walk`, похожим образом является итерируемым, но пример откладывается до главы 20, где будут раскрыты основы этого инструмента — генераторы и `yield`.

Протокол итерации также представляет собой причину помещения некоторых результатов внутрь вызова `list` для вывода одновременно всех их значений. Итерируемые объекты возвращают по одному результату за раз, а не физический список:

```
>>> R = range(5)
>>> R                      # В Python 3.X диапазоны являются итерируемыми объектами
range(0, 5)
>>> I = iter(R)           # Использование протокола итерации для выпуска результатов
>>> next(I)
0
>>> next(I)
1
>>> list(range(5)) # Или применение вызова list для сбора всех результатов
[0, 1, 2, 3, 4]
```

Обратите внимание, что в Python 2.X вызов `list` не требуется (там `range` строит реальный список), и в нем нет необходимости в Python 3.X для контекстов, где итерация происходит автоматически (таких как внутри циклов `for`). Тем не менее, он нужен здесь для отображения значений в Python 3.X и также может требоваться, когда необходимо поведение, подобное спискам, или множественные просмотры для объектов, которые производят результаты по запросу в Python 2.X или 3.X (дополнительные сведения будут даны позже).

Обладая лучшим пониманием протокола итерации, теперь вы должны быть в состоянии объяснить, почему представленный в предыдущей главе инструмент `enumerate` работает именно так, как он работает:

```
>>> E = enumerate('spam')      # Инструмент enumerate также итерируемый
>>> E
<enumerate object at 0x00000000029B7678>
>>> I = iter(E)
>>> next(I)                  # Генерировать результаты с помощью протокола итерации
(0, 's')
>>> next(I)                  # Либо использовать вызов list
                               # для принудительного запуска генерации
(1, 'p')
>>> list(enumerate('spam'))
[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]
```

Обычно данный механизм для нас невидим, т.к. циклы `for` запускают его автоматически для прохода по результатам. На самом деле все, что в Python выполняет просмотр слева направо, задействует протокол итерации тем же самым способом — в том числе инструмент, рассматриваемый в следующем разделе.

Списковые включения: первый подробный взгляд

После выяснения, каким образом работает протокол итерации, давайте теперь перейдем к одному из его самых распространенных сценариев использования. Вместе с циклами `for` списковые включения входят в число наиболее известных контекстов, в которых применяется протокол итерации.

В предыдущей главе было показано, как использовать `range` для изменения списка по мере прохождения через него:

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)):
...     L[i] += 10
...
>>> L
[11, 12, 13, 14, 15]
```

Прием работает, но как там упоминалось, он может не быть оптимальным “рекомендуемым” подходом в Python. В наши дни выражение спискового включения переводит многие ранее применяемые кодовые шаблоны в категорию устаревших. Скажем, мы можем заменить приведенный выше цикл одиночным выражением, которое производит желаемый результирующий список:

```
>>> L = [x + 10 for x in L]
>>> L
[21, 22, 23, 24, 25]
```

Конечный результат аналогичен, но такой прием требует меньшего объема кодирования с нашей стороны и, вероятно, будет выполняться существенно быстрее. Версия со списковым включением не точно соответствует версии с оператором цикла `for`, т.к. она создает *новый* списковый объект (что может иметь значение при наличии множества ссылок на исходный список), но она достаточно близка для большинства приложений и является распространенным и довольно удобным подходом, заслуживающим более детального рассмотрения.

Основы списковых включений

Мы кратко обсуждали списковые включения в главе 4. Их синтаксис происходит из конструкции в системе обозначений теории множеств, которая применяет операцию к каждому элементу внутри множества, но для использования данного инструмента знание теории множеств вовсе не требуется. В Python большинство людей находят, что списковое включение просто выглядит как цикл `for`, повернутый назад.

Чтобы понять синтаксис, разберем пример из предыдущего раздела более подробно:

```
L = [x + 10 for x in L]
```

Списковые включения записываются в квадратных скобках, потому что в конечном итоге представляют собой способ создания нового списка. Они начинаются с произвольно составленного выражения, в котором задействована введенная нами переменная цикла ($x + 10$). За выражением следует то, что теперь вы должны опознать как заголовок цикла `for`, в котором указана переменная цикла и итерируемый объект (`for x in L`).

Для запуска выражения Python выполняет итерацию по `L` внутри интерпретатора, присваивая переменной цикла `x` каждый элемент по очереди, и накапливает результаты прогона элементов через выражение с левой стороны. Получаемый обратно результирующий список является в точности тем, что определяет списковое включение – новый список, содержащий $x + 10$ для каждого x в `L`.

Говоря формально, списковые включения никогда не являются по-настоящему обязательными, т.к. мы всегда можем построить список результатов выполнения выражения вручную с помощью циклов `for`, дополняющих список результатов по мере продвижения:

```
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[31, 32, 33, 34, 35]
```

На самом деле это именно то, что внутренне делает списковое включение.

Однако списковые включения лаконичнее в плане записи, а поскольку такой кодовый шаблон построения результирующих списков настолько часто применяются при работе в Python, они оказываются очень удобными во многих контекстах. Кроме того, в зависимости от версии Python и кода списковые включения могут выполняться гораздо быстрее ручных операторов цикла `for` (часто примерно вдвое быстрее), потому что их итерации функционируют со скоростью кода на языке С внутри интерпретатора, а не вручную написанного кода Python. Использование выражений списковых включений нередко дают значительное преимущество в отношении производительности, особенно для крупных наборов данных.

Использование списковых включений с файлами

Давайте займемся еще одним распространенным приложением списковых включений, чтобы исследовать их более детально. Вспомните, что файловый объект имеет метод `readlines`, который загружает весь файл целиком в список строк:

```
>>> f = open('script2.py')
>>> lines = f.readlines()
>>> lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']
```

Код работает, но все строки в результате содержат в конце символ новой строки (`\n`). Во многих программах символ новой строки становится помехой — нам необходимо позаботиться об отсутствии в выводе перемежающихся пустых строк и т.п. Было бы неплохо иметь возможность избавиться сразу от всех символов новой строки, не так ли?

Всякий раз, когда мы начинаем думать о выполнении какой-то операции над каждым элементом в последовательности, то находимся в сфере влияния списковых включений. Скажем, предполагая сохранение переменной `lines` в том виде, как она была в предыдущем взаимодействии, следующий код делает свою работу, прогоняя каждую строку в списке через строковый метод `rstrip` для удаления пробельных символов с правой стороны (подошел бы и срез `line[:-1]`, но лишь в ситуации, когда мы можем быть уверены в том, что все строки надлежащим образом заканчиваются символом `\n`, а в последней строке файла так может быть не всегда):

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(x ** 32)']
```

Код работает, как было запланировано. Тем не менее, поскольку списковые включения являются итерационным контекстом подобно операторам цикла `for`, мы даже не обязаны заблаговременно открывать файл. Если мы откроем его внутри выражения, тогда списковое включение автоматически будет применять протокол итерации, описанный ранее в главе. То есть оно будет читать по одной строке из файла за раз, вызывая метод `next` файлового объекта, прогонять эту строку через выражение `rstrip`.

и добавлять полученный результат в итоговый список. И снова мы получаем то, что запрашивали – результаты выполнения `rstrip` для всех строк в файле:

```
>>> lines = [line.rstrip() for line in open('script2.py')]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(x ** 32)']
```

Это выражение многое делает неявно, но мы получаем большой объем работы “даром” – Python просматривает файл по строкам и автоматически строит список результатов выполнения операции. Оно также представляет собой эффективный способ кодирования такой операции: из-за того, что большинство работ делается внутри интерпретатора Python, списковое включение может быть быстрее эквивалентного оператора `for` и не загружает все содержимое файла в память подобно ряду других методик. Опять-таки, особенно для крупных файлов, преимущества списковых включений могут оказаться значительными.

Помимо эффективности списковые включения также удивительно выразительные. В рассмотренном примере при проведении итераций мы можем выполнять в отношении строк файла любую строковую операцию. В целях иллюстрации ниже показан эквивалент в виде спискового включения для приведенного ранее примера с итератором файлового объекта, преобразующим строки в верхний регистр, вместе с рядом других типичных операций:

```
>>> [line.upper() for line in open('script2.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
>>> [line.rstrip().upper() for line in open('script2.py')]
['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(X ** 32)']
>>> [line.split() for line in open('script2.py')]
[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(x', '**', '32')']]
>>> [line.replace(' ', '!') for line in open('script2.py')]
['import!sys\n', 'print(sys.path)\n', 'x!=!2\n', 'print(x!**!32)\n']
>>> [('sys' in line, line[:5]) for line in open('script2.py')]
[(True, 'impor'), (True, 'print'), (False, 'x = 2'), (False, 'print')]
```

Вспомните, что выстраивание вызовов методов в *цепочку* во втором примере работает из-за того, что строковые методы возвращают новую строку, к которой можно применять другой строковый метод. В последнем примере показано, что можно также накапливать *множество* результатов, пока они помещены в коллекцию вроде кортежа или списка.



Один тонкий момент: вспомните из главы 9, что файловые объекты сами автоматически закрывают файлы, если на момент обработки сборщиком мусора они все еще открыты. Следовательно, такие списковые включения также будут автоматически закрывать файлы, когда их временные файловые объекты подвергается сборке мусора после того, как выражение выполнено. Однако за пределами CPython при выполнении списковых включений в цикле может возникнуть желание вручную закрывать файлы, чтобы немедленно освобождать ресурсы. Дополнительные сведения ищите в главе 9.

Расширенный синтаксис списковых включений

На практике списковые включения могут быть более развитыми и в своих самых полных формах даже вводить своего рода *мини-язык итераций*. Давайте бегло взглянем на их синтаксические средства.

Конструкции фильтров: `if`

В качестве одного особенно полезного расширения цикл `for`, вложенный в выражение спискового включения, может иметь ассоциированную конструкцию `if`, позволяющую *отфильтровывать* элементы результата, для которых проверка дала ложное значение.

Предположим, что нам необходимо повторить пример с просмотром файла из предыдущего раздела, но накапливать только строки, которые начинаются с буквы `p` (возможно, первый символ в каждой строке определяет код какого-то действия). Задача решается добавлением к выражению конструкции фильтра `if`:

```
>>> lines = [line.rstrip() for line in open('script2.py') if line[0] == 'p']
>>> lines
['print(sys.path)', 'print(x ** 32)']
```

Здесь конструкция `if` проверяет каждую строку, прочитанную из файла, чтобы выяснить, начинается ли она с символа `p`; если нет, тогда строка не включается в результатирующий список. Хотя выражение довольно большое, его легко понять, если перевести в простой эквивалентный оператор цикла `for`. В общем случае мы всегда можем транслировать списковое включение в оператор `for`, добавляя отступы к каждой последующей части:

```
>>> res = []
>>> for line in open('script2.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print(sys.path)', 'print(x ** 32)']
```

Такой оператор `for` работает равнозначно, но он занимает до четырех строк вместо одной и может выполняться медленнее. На самом деле при необходимости в списковое включение можно помещать значительный объем логики. Следующий код работает подобно предыдущему, но выбирает только строки, которые *оканчиваются цифрой* (перед завершающим символом новой строки), выполняя фильтрацию с помощью более сложного выражения в правой части (для файлов с пустыми строками `[-1]` понадобится заменить на `[-1:]`):

```
>>> [line.rstrip() for line in open('script2.py') if line.rstrip()[-1].isdigit()]
['x = 2']
```

Ниже приведен еще один пример фильтра `if`. Первое выражение дает общее количество строк в текстовом файле, а второе отбрасывает пробельные символы в начале и конце, чтобы *не учитывать при подсчете пустые строки* (этот файл, не входящий в состав кода примеров, содержит строки с описанием опечаток, найденных корректором в черновом варианте настоящей книги):

```
>>> fname = r'd:\books\5e\lp5e\draft1typos.txt'                                # Все строки
>>> len(open(fname).readlines())
263
>>> len([line for line in open(fname) if line.strip() != ''])    # Непустые строки
185
```

Вложенные циклы: `for`

При необходимости списковые включения могут становиться даже еще сложнее — например, содержать *вложенные циклы*, записанные в виде серии конструкций `for`. В действительности их полный синтаксис разрешает указывать любое количество конструкций `for`, каждая из которых может иметь необязательную связанную конструкцию `if`.

Скажем, следующее списковое включение строит список результатов конкатенации `x + y` для каждого элемента `x` в одной строке и каждого элемента `y` в другой. Оно фактически собирает все упорядоченные комбинации символов в двух строках:

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

И снова понять такое выражение проще, если преобразовать в форму оператора с добавлением отступов к его частям. Ниже показан эквивалентный, хотя более медленный способ достижения того же результата:

```
>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Но при менее высоком уровне сложности списковые включения часто могут становиться весьма компактными, что говорит, несомненно, в их пользу. В целом они предназначены для простых видов итерации; для более сложной работы структуру с оператором `for` вероятно будет легче понять и модифицировать в будущем. Это обычная ситуация в программировании: если решение оказывается сложным для понимания, то оно, скорее всего, неудачно.

Поскольку списковые включения лучше усваиваются небольшими порциями, мы здесь не будем продолжать историю, а возвратимся к ним в главе 20 в контексте инструментов функционального программирования. Там мы более формально определим их синтаксис и рассмотрим дополнительные примеры. Вы увидите, что списковые включения в такой же степени связаны с *функциями*, как они связаны с *операторами* цикла.



Общая оценка для всех заявлений о производительности в этой книге, касающихся списковых включений или других инструментов: относительная скорость выполнения кода во многом зависит от конкретного тестируемого кода и используемой версии Python, а также предрасположена к изменениям от выпуска к выпуску.

Например, в CPython 2.7 и 3.3 списковые включения по-прежнему могут быть в два раза быстрее соответствующих циклов `for` в одних тестах, но лишь минимально быстрее в других и возможно даже слегка медленнее в случае применения конструкций фильтров `if`.

В главе 21 мы выясним, как измерять время выполнения кода, и научимся интерпретировать файл `listcomp-speed.txt` в пакете примеров для книги, который фиксирует время выполнения кода из этой главы. А пока примите к сведению, что абсолютные показатели в оценках производительности столь же труднодостижимы, как единодушие в проектах с открытым кодом!

Другие итерационные контексты

Позже в книге будет показано, что определяемые пользователем классы также могут реализовывать протокол итерации. По указанной причине иногда важно знать, какие встроенные инструменты их используют — любой инструмент, применяющий протокол итерации, будет автоматически работать с любым встроенным типом или определяемым пользователем классом, который данный протокол поддерживает.

До сих пор итераторы демонстрировались в контексте оператора цикла `for`, т.к. текущая часть книги сосредоточена на операторах. Однако имейте в виду, что любой встроенный инструмент, который просматривает объекты слева направо, задействует протокол итерации. Это относится и к рассмотренным ранее циклам `for`:

```
>>> for line in open('script2.py'): # Использование итератора файлового объекта
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

Но инструментов намного больше. Скажем, списковые включения и встроенная функция `map` используют тот же протокол, что и родственный им цикл `for`. В случае применения к файлу они оба автоматически используют итератор файлового объекта для его просмотра строка за строкой, извлекая итератор с помощью `__iter__` и вызывая по мере продвижения `__next__`:

```
>>> uppers = [line.upper() for line in open('script2.py')]
>>> uppers
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
>>> map(str.upper, open('script2.py')) # В Python 3.X функция map возвращает
                                         # итерируемый объект
<map object at 0x0000000029476D8>
>>> list(map(str.upper, open('script2.py')))
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
```

Вызов `map` был кратко представлен в предыдущей главе (и мимоходом в главе 4); этот встроенный инструмент применяет вызов функции к каждому элементу в переданном итерируемом объекте. Инструмент `map` похож на списковое включение, но он более ограничен, т.к. требует функцию, а не произвольное выражение. Он также *возвращает* итерируемый объект в Python 3.X, так что для вывода сразу всех значений его придется поместить внутрь вызова `list`; более подробно об указанном изменении речь пойдет позже в главе. Поскольку встроенная функция `map` подобно списковому включению имеет отношение к циклам `for` и функциям, мы продолжим ее исследование в главах 19 и 20.

Многие другие встроенные функции Python также обрабатывают итерируемые объекты. Например, `sorted` сортирует элементы в итерируемом объекте; `zip` объединяет элементы из итерируемых объектов; `enumerate` формирует пары из элементов в итерируемом объекте и их относительных позиций; `filter` выбирает элементы из итерируемого объекта, для которых заданная функция дает истинное значение; `reduce` прогоняет пары элементов в итерируемом объекте через указанную функцию. Все они принимают итерируемые объекты, а `zip`, `enumerate` и `filter` подобно `map` также *возвращают* итерируемый объект в Python 3.X. Ниже перечисленные инструменты

показаны в действии, автоматически запуская итератор файлового объекта, чтобы читать файл строка за строкой:

```
>>> sorted(open('script2.py'))
['import sys\n', 'print(sys.path)\n', 'print(x ** 32)\n', 'x = 2\n']
>>> list(zip(open('script2.py'), open('script2.py')))
[('import sys\n', 'import sys\n'), ('print(sys.path)\n', 'print(sys.path)\n'),
 ('x = 2\n', 'x = 2\n'), ('print(x ** 32)\n', 'print(x ** 32)\n')]
>>> list(enumerate(open('script2.py')))
[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'),
 (3, 'print(x ** 32)\n')]
>>> list(filter(bool, open('script2.py')))           # nonempty=True
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']
>>> import functools, operator
>>> functools.reduce(operator.add, open('script2.py'))
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'
```

Все они являются итерационными инструментами, но исполняют индивидуальные роли. Мы встречали `zip` и `enumerate` в предыдущей главе, а `filter` и `reduce` будут обсуждаться в главе 19 при рассмотрении области функционального программирования, поэтому мы отложим выяснение деталей до указанной главы. Здесь важно отметить лишь то, что они используют протокол итерации для файлов и других итерируемых объектов.

Впервые мы видели функцию `sorted` в работе в главе 4, и она применялась для словарей в главе 8. `sorted` является встроенной функцией, которая задействует протокол итерации – она похожа на первоначальный списковый метод `sort`, но возвращает в качестве результата новый отсортированный список и выполняется на любом итерируемом объекте. Обратите внимание, что в отличие от `map` и остальных функций `sorted` в Python 3.X возвращает реально существующий *список*, а не итерируемый объект.

Интересен тот факт, что протокол итерации в современном Python даже еще более вездесущий, чем демонстрировалось на примерах до сих пор – по существу абсолютно все встроенные инструменты в наборе Python, которые просматривают объект слева направо, определены для использования протокола итерации на указанном объекте. Сюда даже входят такие инструменты, как встроенные функции `list` и `tuple` (которые строят новые объекты из итерируемых объектов) и строковый метод `join` (который создает новую строку путем помещения подстроки между строками, содержащимися в итерируемом объекте). Следовательно, они также будут работать на открытом файле и автоматически читать по одной строке за раз:

```
>>> list(open('script2.py'))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']
>>> tuple(open('script2.py'))
('import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n')
>>> '&&'.join(open('script2.py'))
'import sys\n&&print(sys.path)\n&&x = 2\n&&print(x ** 32)\n'
```

К этой категории относятся даже такие инструменты, которые вы могли не ожидать. Скажем, присваивание последовательностей, проверка членства `in`, присваивание срезов и списковый метод `extend` также задействуют протокол итерации для просмотра и потому автоматически читают файл по строкам:

```

>>> a, b, c, d = open('script2.py')      # Присваивание последовательностей
>>> a, d
('import sys\n', 'print(x ** 32)\n')
>>> a, *b = open('script2.py')          # Расширенная форма
>>> a, b
('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n'])
>>> 'y = 2\n' in open('script2.py')     # Проверка членства
False
>>> 'x = 2\n' in open('script2.py')
True

>>> L = [11, 22, 33, 44]                # Присваивание срезов
>>> L[1:3] = open('script2.py')
>>> L
[11, 'import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n', 44]
>>> L = [11]
>>> L.extend(open('script2.py'))        # Метод list.extend
>>> L
[11, 'import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

```

Согласно главе 8 метод `extend` выполняет итерацию автоматически, но метод `append` – нет; применяйте последний (или похожий метод) для добавления итерируемого объекта в список без итерации с потенциальной итерацией в более позднее время:

```

>>> L = [11]
>>> L.append(open('script2.py'))    # Метод list.append не выполняет итерацию
>>> L
[11, <_io.TextIOWrapper name='script2.py' mode='r' encoding='cp1252'>]
>>> list(L[1])
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

```

Итерация является широко поддерживаемой и мощной моделью. Ранее мы видели, что встроенный вызов `dict` тоже принимает итерируемый результат `zip` (см. главы 8 и 13). То же самое касается вызова `set`, а также более новых выражений включения множеств и словарей в Python 3.X и 2.7, которые встречались в главах 4, 5 и 8:

```

>>> set(open('script2.py'))
{'print(x ** 32)\n', 'import sys\n', 'print(sys.path)\n', 'x = 2\n'}
>>> {line for line in open('script2.py')}
{'print(x ** 32)\n', 'import sys\n', 'print(sys.path)\n', 'x = 2\n'}
>>> {ix: line for ix, line in enumerate(open('script2.py'))}
{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(x ** 32)\n'}

```

На самом деле включения множеств и словарей поддерживают расширенный синтаксис списковых включений, который мы обсуждали выше в главе, в том числе проверки `if`:

```

>>> {line for line in open('script2.py') if line[0] == 'p'}
{'print(x ** 32)\n', 'print(sys.path)\n'}
>>> {ix: line for (ix, line) in enumerate(open('script2.py')) if line[0] == 'p'}
{1: 'print(sys.path)\n', 3: 'print(x ** 32)\n'}

```

Подобно списковому включению оба выражения просматривают файл строка за строкой и отбирают строки, начинающиеся с буквы `p`. В конечном итоге они также

строят множества и словари, но мы получаем много работы “даром”, комбинируя синтаксис итерации файлов и включений. Позже в книге встретится родственный включениям инструмент – генераторные выражения, которые используют тот же самый синтаксис и работают на итерируемых объектах, но в добавок сами являются итерируемыми:

```
>>> list(line.upper() for line in open('script2.py'))    # См. главу 20
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
```

Другие встроенные функции также поддерживают протокол итерации, но, к сожалению, некоторые из них труднее задействовать в интересных примерах, имеющих отношение к файлам! Например, `sum` вычисляет сумму всех чисел в любом итерируемом объекте; `any` и `all` возвращают `True`, если соответственно любой или все элементы в итерируемом объекте истинны, а `max` и `min` возвращают соответственно наибольший или наименьший элемент в итерируемом объекте. Подобно `reduce` все инструменты в приведенных далее примерах принимают в аргументе любой итерируемый объект и применяют протокол итерации для его просмотра, но возвращают одиночный результат:

```
>>> sum([3, 2, 4, 1, 5, 0])      # sum ожидает только числа
15
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
>>> max([3, 2, 5, 1, 4])
5
>>> min([3, 2, 5, 1, 4])
1
```

Строго говоря, функции `max` и `min` могут также применяться к файлам – они автоматически используют протокол итерации для просмотра файла и отбора строк соответственно с наибольшим или наименьшим строковым значением (хотя я оставляю допустимые сценарии применения целиком на ваше воображение):

```
>>> max(open('script2.py'))    # Стока с максимальным/минимальным
                                # строковым значением
'x = 2\n'
>>> min(open('script2.py'))
'import sys\n'
```

Есть еще последний итерационный контекст, заслуживающий упоминания, пусть даже и краткого: в главе 18 будет показано, что в вызовах функций можно применять специальную форму `*аргумент` для распаковки коллекций значений в индивидуальные аргументы. Как вы вероятно уже можете предугадать, здесь подойдет также любой итерируемый объект, в том числе и файлы (ищите в главе 18 дополнительные сведения по такому синтаксису вызовов, в главе 20 раздел, в котором эта идея распространяется на генераторные выражения, и в главе 11 советы по использованию следующего оператора `print` из Python 3.X обычным образом в Python 2.X):

```
>>> def f(a, b, c, d): print(a, b, c, d, sep='&')
...
>>> f(1, 2, 3, 4)
1&2&3&4
>>> f(*[1, 2, 3, 4])           # Распаковывает в аргументы
1&2&3&4
```

```
>>>
>>> f(*open('script2.py'))      # Так же выполняет итерацию по строкам!
import sys
&print(sys.path)
&x = 2
&print(x ** 32)
```

В действительности из-за того, что данный синтаксис распаковки аргументов в вызовах принимает итерируемые объекты, становится возможным также применение встроенной функции `zip` для *развертывания* сжатых кортежей, делая предшествующие или вложенные результаты `zip` аргументами для другого вызова `zip` (предостережение: вам вряд ли стоит изучать следующий пример, если вам предстоит вскоре заняться тяжеловесными механизмами!):

```
>>> X = (1, 2)
>>> Y = (3, 4)
>>>
>>> list(zip(X, Y))           # Сжатие кортежей: возвращает итерируемый объект
[(1, 3), (2, 4)]
>>>
>>> A, B = zip(*zip(X, Y))    # Развертывание сжатых кортежей!
>>> A
(1, 2)
>>> B
(3, 4)
```

Еще несколько инструментов в Python, такие как встроенная функция `range` и объекты словарных представлений, *возвращают* итерируемые объекты вместо того, чтобы обрабатывать их. В следующем разделе вы увидите, каким образом они задействуют протокол итерации в Python 3.X.

Новые итерируемые объекты в Python 3.X

Одним из фундаментальных отличий линейки Python 3.X является более сильный акцент на итераторах, чем в Python 2.X. Наряду с моделью Unicode и обязательными классами нового стиля это представляет собой одно из самых радикальных изменений Python 3.X.

В частности, вдобавок к итераторам, ассоциированным со встроенными типами вроде файлов и словарей, словарные методы `keys`, `values` и `items` в Python 3.X возвращают итерируемые объекты, как и встроенные функции `range`, `map`, `zip` и `filter`. В предыдущем разделе было показано, что последние три функции не только возвращают итерируемые объекты, но и обрабатывают их. Все инструменты подобного рода в Python 3.X производят результаты по запросу, а не создают списки результатов, как было в Python 2.X.

Влияние на код Python 2.X: доводы за и против

Хотя все это приводит к экономии пространства памяти, оно может повлиять на стиль написания кода в ряде контекстов. Скажем, до сих пор в разных местах книги мы должны были помещать результаты вызовов некоторых функций и методов внутрь вызова `list(...)`, чтобы заставить их выдать все результаты сразу для *отображения*:

```
>>> zip('abc', 'xyz') # Итерируемый объект в Python 3.X (список в Python 2.X)
<zip object at 0x00000000294C308>
```

```
>>> list(zip('abc', 'xyz'))      # Принудительная выдача списка результатов
                                         # для отображения в Python 3.X
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Похожее преобразование требуется, когда мы хотим применять списковые операции или *операции над последовательностями* к большинству итерируемых объектов, которые генерируют элементы по запросу – для индексации, нарезания или конкатенации самого итерируемого объекта, например. Списковые результаты для таких инструментов в Python 2.X поддерживают операции подобного рода напрямую:

```
>>> Z = zip((1, 2), (3, 4))      # В отличие от списков Python 2.X нельзя
                                         # индексировать и т.д.
>>> Z[0]
TypeError: 'zip' object is not subscriptable
Ошибка типа: объект zip не допускает индексацию
```

Как будет более подробно обсуждаться в главе 20, преобразование в списки может также не настолько очевидно требоваться в целях поддержки множественных итераций для по-новому работающих итерационных инструментов, которые предусматривают только один просмотр, скажем, map и zip. В отличие от списковых форм Python 2.X их значения в Python 3.X израсходуются после одиночного прохода:

```
>>> M = map(lambda x: 2 ** x, range(3))
>>> for i in M: print(i)
...
1
2
4
>>> for i in M: print(i)          # В отличие от списков Python 2.X только
                                         # один проход (zip тоже)
...
>>>
```

Такое преобразование в Python 2.X не требуется, потому что функции вроде zip возвращают списки результатов. Тем не менее, в Python 3.X они возвращают итерируемые объекты, производя результат по запросу. В итоге работа кода Python 2.X может быть нарушена, а для отображения в интерактивной подсказке (и возможно в ряде других контекстов) потребуется дополнительный набор, но в более крупных программах это полезное качество – отложенная оценка подобного рода сберегает память и позволяет избежать пауз, связанных с вычислением больших результирующих списков. Давайте посмотрим на некоторые новые итерируемые объекты Python 3.X в действии.

Итерируемый объект `range`

Мы исследовали базовое поведение встроенной функции range в предыдущей главе. В Python 3.X она возвращает итерируемый объект, который генерирует числа в диапазоне по запросу вместо построения результирующего списка в памяти. Это напоминает старую функцию xrange из Python 2.X (см. врезку “Примечание, касающееся нестыковки версий” далее в главе), и вы должны использовать list(range(...)), чтобы получить фактический список, когда он необходим (например, для отображения результатов):

```
C:\code> c:\python33\python
>>> R = range(10)                 # range возвращает итерируемый объект, а не список
>>> R
range(0, 10)
```

```
>>> I = iter(R)          # Создание итератора из итерируемого объекта range
>>> next(I)             # Продвижение на следующий результат
0                      # Что происходит в циклах for, включениях и т.д.
>>> next(I)
1
>>> next(I)
2

>>> list(range(10))    # Принудительное преобразование в список, если необходимо
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

В отличие от списка, возвращаемого таким вызовом в Python 2.X, объекты `range` в Python 3.X поддерживают только итерацию, индексацию и функцию `len`. Они не поддерживают любые другие операции над последовательностями (применяйте `list(...)`, если требуется больше списковых инструментов):

```
>>> len(R)              # range также поддерживает функцию len и индексацию,
                        # но ничего другого
10
>>> R[0]
0
>>> R[-1]
9

>>> next(I)             # Продолжает получать из итератора с оставленного ранее места
3
>>> I.__next__()        # .next() становится __next__(),
                        # но использует новый метод next()
4
```



Примечание, касающееся нестыковки версий. Как уже упоминалось в предыдущей главе, в Python 2.X также имеется встроенная функция по имени `xrange`, которая похожа на `range`, но производит элементы по запросу вместо построения полного списка результатов в памяти. Поскольку это то, что в частности делает новая функция `range` на основе итераторов в Python 3.X, `xrange` больше не доступна в Python 3.X – она была заменена. Однако вы все еще можете встречать и использовать ее в коде Python 2.X, особенно из-за того, что функция `range` там строит результирующий список и потому не настолько эффективна в плане расхода памяти.

Как отмечалось в предыдущей главе, метод `file.readlines()`, применяемый для минимизации занимаемого пространства памяти в Python 2.X, по аналогичным причинам был изъят из Python 3.X в пользу итераторов файловых объектов.

Итерируемые объекты `map`, `zip` и `filter`

Подобно `range` ради экономии пространства памяти встроенные функции `map`, `zip` и `filter` в Python 3.X также стали итерируемыми объектами вместо того, чтобы производить результирующий список целиком в памяти. Все три функции не только обрабатывают итерируемые объекты, как в Python 2.X, но и возвращают итерируемые результаты в Python 3.X. Тем не менее, в отличие от `range` они являются собственными итераторами – результаты после однократного прохода по ним израсходуются. Другими словами, нельзя иметь множество итераторов на их результатах, которые поддерживают разные позиции в них.

Ниже представлен сценарий для встроенной функции `map`, которую мы встречали в предыдущей главе. Как и с другими итерируемыми объектами, при необходимости с помощью `list(...)` можно принудительно получить список, но стандартное поведение позволяет сэкономить значительный объем памяти в случае крупных результирующих наборов:

```
>>> M = map(abs, (-1, 0, 1)) # map возвращает итерируемый объект, а не список
>>> M
<map object at 0x000000000029B75C0>
>>> next(M)      # Использование итератора вручную: результаты израсходуются
1                         # len() или индексация не поддерживается
>>> next(M)
0
>>> next(M)
1
>>> next(M)
StopIteration
>>> for x in M: print(x)      # Итератор map теперь пуст: только один проход
...
>>> M = map(abs, (-1, 0, 1)) # Создать новый итерируемый объект/итератор для
                           # повторного просмотра
>>> for x in M: print(x) # Итерационные контексты автоматически вызывают next()
...
1
0
1
>>> list(map(abs, (-1, 0, 1))) # При необходимости можно принудительно
                               # преобразовать в список
[1, 0, 1]
```

Встроенная функция `zip`, представленная в предыдущей главе, сама по себе является итерационным контекстом, но также возвращает итерируемый объект с итератором, который работает таким же образом:

```
>>> Z = zip((1, 2, 3), (10, 20, 30))    # zip тоже итератор с одним проходом
>>> Z
<zip object at 0x00000000002951108>
>>> list(Z)
[(1, 10), (2, 20), (3, 30)]
>>> for pair in Z: print(pair) # После одного прохода результаты израсходованы
...
>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> for pair in Z: print(pair)      # Итератор используется автоматически
                               # или вручную
...
(1, 10)
(2, 20)
(3, 30)
>>> Z = zip((1, 2, 3), (10, 20, 30))    # Ручная итерация (iter() не требуется)
>>> next(Z)
(1, 10)
>>> next(Z)
(2, 20)
```

Встроенная функция `filter`, которая кратко упоминалась в главе 12, и будет подробно рассматриваться в следующей части книги, аналогична. Она возвращает элементы в итерируемом объекте, для которых переданная функция дает `True` (как уже известно, в Python значение `True` относится к непустым объектам, а `bool` возвращает значение истинности объекта):

```
>>> filter(bool, ['spam', '', 'ni'])
<filter object at 0x00000000029B7B70>
>>> list(filter(bool, ['spam', '', 'ni']))
['spam', 'ni']
```

Подобно большинству инструментов, обсуждаемых в этом разделе, встроенная функция `filter` как *принимает* итерируемый объект для обработки, так и *возвращает* итерируемый объект для генерации результатов в Python 3.X. Обычно ее можно эмулировать посредством расширенного синтаксиса списковых включений, который автоматически проверяет значения истинности:

```
>>> [x for x in ['spam', '', 'ni'] if bool(x)]
['spam', 'ni']
>>> [x for x in ['spam', '', 'ni'] if x]
['spam', 'ni']
```

Итераторы с множеством проходов или с одним проходом

Важно видеть, чем объект, возвращаемый `range`, отличается от объектов, которые возвращаются встроенными функциями, описанными в настоящем разделе. Он поддерживает функцию `len` и индексацию, сам не является итератором (итератор создается вручную с помощью `iter`) и допускает использование в отношении результата множества итераторов с запоминанием их позиций независимым образом:

```
>>> R = range(3)           # range допускает множество итераторов
>>> next(R)
TypeError: range object is not an iterator
Ошибка типа: объект range не является итератором

>>> I1 = iter(R)
>>> next(I1)
0
>>> next(I1)
1
>>> I2 = iter(R)          # Два итератора на одном диапазоне
>>> next(I2)
0
>>> next(I1)              # I1 находится не в том же месте, что и I2
2
```

Напротив, в Python 3.X встроенные функции `zip`, `map` и `filter` не поддерживают множество активных итераторов на том же самом результате. По указанной причине вызов `iter` необязателен для прохода по результатам таких объектов – они сами являются `iter` (в Python 2.X упомянутые встроенные функции возвращают списки с возможностью множества проходов, так что следующий подход к ним неприменим):

```
>>> Z = zip((1, 2, 3), (10, 11, 12))
>>> I1 = iter(Z)
>>> I2 = iter(Z)          # Два итератора на одном результате zip
>>> next(I1)
(1, 10)
```

```

>>> next(I1)
(2, 11)
>>> next(I2)          # (Python 3.X) I2 находится не в том же месте, что и I1!
(3, 12)

>>> M = map(abs, (-1, 0, 1))    # То же самое для результата map (и filter)
>>> I1 = iter(M); I2 = iter(M)
>>> print(next(I1), next(I1), next(I1))
1 0 1
>>> next(I2)          # (Python 3.X) Одиночный проход исчерпал элементы!
StopIteration

>>> R = range(3)      # Но результат range допускает множество итераторов
>>> I1, I2 = iter(R), iter(R)
>>> [next(I1), next(I1), next(I1)]
[0 1 2]
>>> next(I2)          # Множество активных проходов подобно спискам Python 2.X
0

```

Когда мы будем писать код собственных итерируемых объектов с помощью классов позже в книге (в главе 30), то заметим, что множество итераторов обычно поддерживается за счет возвращения новых объектов для вызова `iter`; одиночный итератор, как правило, означает возвращение объектом самого себя. В главе 20 также обнаружится, что *генераторные функции и выражения* в этом отношении ведут себя подобно `map` и `zip`, а не `range`, поддерживая только один активный проход. Там мы увидим ряд тонких последствий итераторов с одним проходом в циклах, пытающихся выполнить проход несколько раз – код, который раньше обходился с такими объектами как со списками, потерпит неудачу без ручного преобразования их в списки.

Итерируемые словарные представления

Наконец, как было кратко показано в главе 8, словарные методы `keys`, `values` и `items` в Python 3.X возвращают итерируемые объекты *представлений*, которые генерируют элементы результата по одному за раз вместо выработки полных списков результатов в памяти. Представления также доступны в Python 2.7 как вариант, но со специальными именами методов во избежание влияния на существующий код. Элементы представлений поддерживают такое же физическое упорядочение, как у словаря, и отражают изменения, вносимые в лежащий в основе словарь. Теперь, зная больше об итерируемых объектах, рассмотрим остаток истории – в Python 3.3 (порядок следования ключей у вас может быть другим):

```

>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'b': 2, 'c': 3}

>>> K = D.keys()          # Объект представления в Python 3.X, не список
>>> K
dict_keys(['a', 'b', 'c'])

>>> next(K)              # Представления сами по себе не являются итераторами
TypeError: dict_keys object is not an iterator
Ошибка типа: объект dict_keys не является итератором

>>> I = iter(K)          # Итерируемые объекты представлений имеют итератор,
                           # который может использоваться вручную,
                           # но не поддерживают len() и индексацию
>>> next(I)
'a'
'b'

```

```
>>> for k in D.keys(): print(k, end=' ') # Все итерационные контексты
                                         # применяют итераторы автоматически
...
a b c
```

Как и со всеми итерируемыми объектами, производящими значения по запросу, вы всегда можете заставить словарное представление Python 3.X строить реальный список, передавая его встроенной функции `list`. Однако поступать так обычно не требуется, кроме как для отображения результатов в интерактивной подсказке или для применения списковых операций вроде индексации:

```
>>> K = D.keys()
>>> list(K)      # По-прежнему можно при необходимости создать реальный список
['a', 'b', 'c']

>>> V = D.values()    # То же самое касается представлений values() и items()
>>> V
dict_values([1, 2, 3])
>>> list(V) # Для отображения либо индексации как списка необходим вызов list()
[1, 2, 3]

>>> V[0]
TypeError: 'dict_values' object does not support indexing
Ошибка типа: объект dict_values не поддерживает индексацию
>>> list(V)[0]
1

>>> list(D.items())
[('a', 1), ('b', 2), ('c', 3)]

>>> for (k, v) in D.items(): print(k, v, end=' ')
...
a 1 b 2 c 3
```

Вдобавок словари Python 3.X по-прежнему сами являются итерируемыми, имея итератор, который возвращает следующие один за другим ключи. Таким образом, в этом контексте вызывать `keys` напрямую требуется нечасто:

```
>>> D                      # Словари все еще производят итератор
{'a': 1, 'b': 2, 'c': 3}      # На каждой итерации возвращает очередной ключ
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'b'

>>> for key in D: print(key, end=' ')    # По-прежнему нет необходимости
                                         # вызывать keys() для итерации
...
                                         # Но keys также является итерируемым объектом в Python 3.X!
a b c
```

Запомните еще раз, что поскольку `keys` больше не возвращает список, традиционный кодовый шаблон для просмотра словаря по отсортированным ключам в Python 3.X работать не будет. Взамен сначала понадобится преобразовать представления ключей посредством вызова `list` или использовать вызов `sorted` либо на представлении ключей, либо на самом словаре, как показано ниже. Мы видели это в главе 8, но данный аспект достаточно важен для программистов на Python 2.X, чтобы продемонстрировать его снова:

```
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> for k in sorted(D.keys()): print(k, D[k], end=' ')
...
a 1 b 2 c 3
>>> for k in sorted(D): print(k, D[k], end=' ')      # "Рекомендуемая"
# сортировка ключей
...
a 1 b 2 c 3
```

Другие темы, связанные с итерацией

Как упоминалось в начале главы, списковые включения и итерируемые объекты будут дополнительно рассматриваться в главе 20 в сочетании с функциями, а также в главе 30 во время исследования классов. Позже вы увидите, что:

- определяемые пользователем функции можно превращать в генераторные функции с помощью операторов `yield`;
- списковые включения трансформируются в итерируемые генераторные выражения, когда записываются в круглых скобках;
- определяемые пользователем классы делаются итерируемыми посредством перегрузки операций `__iter__` или `__getitem__`.

В частности, итерируемые объекты, определяемые пользователем, описываются классами и позволяют использовать произвольные объекты и операции в любых итерационных контекстах, которые встречались ранее в главе. За счет поддержки единственной операции — *итерации* — объекты могут эксплуатироваться в широком многообразии контекстов и инструментов.

Резюме

В главе были исследованы концепции, связанные с циклами в Python. Мы впервые достаточно пристально взглянули на *протокол итерации* в Python как способ участия в итерационных циклах для объектов, не являющихся последовательностями, и на *списковые включения*. Мы выяснили, что списковое включение представляет собой выражение, похожее на цикл `for`, который применяет другое выражение ко всем элементам в любом итерируемом объекте. Попутно мы также увидели в работе прочие встроенные итерационные инструменты и ознакомились с последними дополнениями Python 3.X, касающимися итерации.

На этом наш тур по индивидуальным процедурным операторам и связанным инструментам завершен. Следующая глава закончит текущую часть книги обсуждением вариантов документирования кода Python. Хотя документация несколько отступает от более детальных аспектов написания кода, она также входит в состав общей синтаксической модели и считается важным компонентом хорошо написанных программ. В следующей главе также будет предложен набор упражнений для данной части книги, которые имеет смысл проработать до перехода к рассмотрению более крупных структур, таких как функции. Но прежде чем двигаться дальше, закрепите полученные знания, ответив на традиционные контрольные вопросы главы.

Проверьте свои знания: контрольные вопросы

1. Как связаны циклы `for` и итерируемые объекты?
2. Как связаны циклы `for` и списковые включения?
3. Назовите четыре итерационных контекста в языке Python.
4. Каков наилучший способ чтения текстового файла строка за строкой на сегодняшний день?
5. Что за оружие вы ожидали бы увидеть у испанской инквизиции?

Проверьте свои знания: ответы

1. Цикл `for` использует *протокол итерации* для прохода по элементам в итерируемом объекте, который участвует в итерации. Сначала он извлекает итератор из итерируемого объекта, передавая объект встроенной функции `iter`, а затем на каждой итерации вызывает метод `_next__` объекта итератора в Python 3.X и перехватывает исключение `StopIteration` для определения, когда останавливать выполнение цикла. В Python 2.X метод называется `next` и запускается встроенной функцией `next` в Python 3.X и 2.X. Любой объект, который поддерживает такую модель, будет работать в цикле `for` и во всех остальных итерационных контекстах. Для ряда объектов, которые сами по себе являются итераторами, начальный вызов `iter` считается излишним, но безвредным.
2. Оба представляют собой итерационные инструменты и контексты. Списковые включения – это лаконичный и зачастую эффективный способ выполнения общей задачи цикла `for`: накопление результатов применения выражения ко всем элементам в итерируемом объекте. Списковое включение всегда возможно транслировать в цикл `for`, а часть выражения спискового включения синтаксически выглядит похожей на заголовок цикла `for`.
3. К итерационным контекстам в Python относятся цикл `for`, списковые включения, встроенная функция `map`, выражение проверки членства `in`, а также встроенные функции `sorted`, `sum`, `any` и `all`. В эту категорию также входят встроенные функции `list` и `tuple`, строковые методы `join` и присваивания последовательностей – все они используют протокол итерации (см. ответ на вопрос 1) для прохода по итерируемым объектам по одному элементу за раз.
4. Наилучший на сегодняшний день способ чтения строк из текстового файла – вообще не читать его явно. Взамен откройте файл внутри итерационного контекста, такого как цикл `for` или списковое включение, и позвольте итерационному инструменту просматривать по одной строке за раз, выполняя на каждой итерации метод `next` файлового объекта. Такой подход, как правило, лучше в плане простоты кода, расхода памяти и возможно требований к скорости выполнения.
5. Я приму любой из следующих правильных ответов: запугивание, устрашение, элегантная красная униформа, удобное кресло и мягкие подушки.

Документация

Текущая часть книги завершается рассмотрением методик и инструментов, используемых для документирования кода Python. Несмотря на то что код Python задуман быть крайне читабельным, небольшое число удачно размещенных доступных для человека комментариев может во многом содействовать лучшему пониманию особенностей работы ваших программ другими. Вы увидите, что Python содержит и синтаксис, и инструменты, призванные сделать процесс документирования как можно более легким. В частности, раскрываемая в главе система *PyDoc* способна визуализировать внутреннюю документацию модуля либо в виде простого текста в оболочке, либо в виде HTML-страницы в веб-браузере.

Хотя данная тема больше относится к концепции инструментов, она представлена здесь отчасти из-за того, что в ней затрагивается синтаксическая модель Python, и отчасти как ресурс для читателей, старающихся понять инструментальный набор Python. Для последней цели будут расширены указания по документированию, впервые предложенные в главе 4. Как обычно, поскольку текущая глава завершает первую часть книги, в конце помимо контрольных вопросов по итогам главы приводится ряд предостережений относительно распространенных ловушек и набор упражнений для пройденной части книги.

Источники документации Python

К этому месту в книге, вероятно, вы уже начинаете осознавать, что Python поступает с опшеломляющим количеством готовой функциональности – встроенных функций и исключений, предварительно определенных атрибутов и методов объектов, стандартных библиотечных модулей и многоного другого. И на самом деле мы лишь слегка коснулись поверхности каждой категории.

Один из первых вопросов, который задают сбитые с точку новички, часто формулируется так: каким образом найти информацию обо всех встроенных инструментах? В настоящем разделе предлагаются советы относительно разнообразных источников документации, доступных в Python. Здесь также представлены *строки документации* и система *PyDoc*, которая их употребляет. Указанные темы кое в чем второстепенны по сравнению с основным языком, но они становятся необходимым знанием, как только ваш код достигнет уровня примеров и упражнений в данной части книги.

Как подытожено в табл. 15.1, отыскать информацию по Python можно во многих местах, как правило, с растущей степенью детализации. Из-за того, что документация является настолько важным инструментом при практическом программировании, в последующих разделах мы исследуем каждую из приведенных категорий.

Таблица 15.1. Источники документации Python

Форма	Роль
Комментарии #	Внутрифайловая документация
Функция dir	Списки атрибутов, доступных в объектах
Строки документации: __doc__	Внутрифайловая документация, присоединенная к объектам
PyDoc: функция help	Интерактивная справка для объектов
PyDoc: отчеты HTML	Документация по модулям, отображающаяся в браузере
Сторонний инструмент Sphinx	Обогащенная документация для более крупных проектов
Стандартный набор руководств	Официальные описания языка и библиотек
Веб-ресурсы	Онлайневые руководства, примеры и т.д.
Печатные издания	Коммерчески совершенные справочники

Комментарии

Как уже известно, комментарии # являются наиболее базовым способом документирования кода. Python просто игнорирует весь текст, следующий за символом # (до тех пор, пока он не находится внутри строкового литерала), поэтому вы можете помечать после # любые слова и описания, имеющие смысл для программистов. Однако такие комментарии доступны только в файлах исходного кода; для написания более широко доступных комментариев понадобится применять строки документации.

На самом деле установившаяся в текущее время практика в целом предполагает, что строки документации лучше подходят для более широкого документирования функциональности (скажем, “файл делает то-то и то-то”), а использование комментариев # лучше ограничивать документированием кода (например, “это странное выражение делает вот что”) и областью действия оператора или небольшой группы операторов внутри сценария либо функции. Вскоре мы вернемся к строкам документации, но сначала давайте посмотрим, каким образом исследовать объекты.

Функция dir

Как было показано ранее,строенная функция dir предоставляет легкий способ получения перечня всех атрибутов, доступных внутри объекта (т.е. его методов и более простых элементов данных). Ее можно вызывать без аргументов, чтобы получить список переменных в области видимости вызывающего кода. Но более удобно то, что dir можно вызывать на любом объекте, имеющем атрибуты, в числе которых импортированные модули и встроенные типы, а также имена типов данных. Скажем, для выяснения, что доступно в module, подобном sys из стандартной библиотеки, его следует импортировать и передать dir:

```
>>> import sys  
>>> dir(sys)  
['__displayhook__', ...остальные имена не показаны..., 'winver']
```

Результаты получены в версии Python 3.7, и большинство имен здесь не показано, т.к. в разных местах они слегка варьируются; чтобы получить полный перечень, выполните приведенные операторы самостоятельно. Фактически в sys имеется 88 атрибутов, хотя в большинстве случаев нас будут интересовать только 77 из них, которые

не начинаются с двух подчеркиваний (их наличие, как правило, означает связь с интерпретатором), или 69 имен, вообще не содержащих ведущие подчеркивания (одно подчеркивание обычно подразумевает неформальную закрытую реализацию) – яркий пример работы спискового включения, обсуждаемого в предыдущей главе:

```
>>> len(dir(sys))                                # Количество имен в sys
88
>>> len([x for x in dir(sys) if not x.startswith('__')]) # Только имена без
                                                       # __ в своем начале
77
>>> len([x for x in dir(sys) if not x[0] == '_']) # Имена без ведущего
                                                       # подчеркивания
69
```

Чтобы выяснить, какие атрибуты предоставляют объекты *встроенных типов*, запустите `dir` с литералом или существующим экземпляром желаемого типа. Например, для просмотра атрибутов списка и строки можно передать пустые объекты:

```
>>> dir([])
['__add__', '__class__', '__contains__', ...more..., 'append', 'clear', 'copy',
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir('')
['__add__', '__class__', '__contains__', ...more..., 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Результат вызова `dir` для любого встроенного типа включает набор атрибутов, которые относятся к реализации данного типа (формально методов перегрузки операций). Почти как в модулях все они начинаются и заканчиваются двумя подчеркиваниями, что делает их особыми, и в этом месте книги вы можете их благополучно игнорировать (они применяются в объектно-ориентированном программировании). Скажем, список имеет 46 атрибутов, но только 11 из них соответствуют именованным методам:

```
>>> len(dir([])), len([x for x in dir([]) if not x.startswith('__')])
(46, 11)
>>> len(dir('')), len([x for x in dir('') if not x.startswith('__')])
(77, 44)
```

Чтобы отфильтровать элементы с двумя подчеркиваниями, которые не представляют обычного интереса для программ, выполните те же самые списковые включения, но с выводом атрибутов. Например, вот именованные атрибуты в списках и словарях Python 3.7:

```
>>> [a for a in dir(list) if not a.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> [a for a in dir(dict) if not a.startswith('__')]
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

Может показаться, что для получения списка атрибутов приходится набирать слишком много кода, но в начале следующей главы будет показано, как помещать такой код в импортируемую и многократно используемую функцию, поэтому набирать его повторно не понадобится:

```
>>> def dir1(x): return [a for a in dir(x) if not a.startswith('__')]
# См. часть IV
```

```
...
>>> dir1(tuple)
['count', 'index']
```

Обратите внимание, что получать перечни атрибутов встроенных типов можно и за счет передачи функции `dir` имени типа вместо литерала:

```
>>> dir(str) == dir('')      # Имя типа и литерал дают тот же результат
>>> dir(list) == dir([])
True
```

Прием работает, т.к. имена вроде `str` и `list`, которые когда-то были функциями преобразования, теперь являются именами типов в Python; обращение к одному из них вызывает его конструктор для создания экземпляра данного типа. Мы обсудим конструкторы и методы перегрузки операций в части VI, когда будем рассматривать классы.

Функция `dir` служит своего рода кратким напоминанием – она предоставляет список имен атрибутов, но ничего не сообщает о том, что имена означают. Для получения дополнительной информации нам нужно перейти к следующему источнику документации.



Определенные IDE-среды для работы с Python, включая IDLE, располагают средствами, которые автоматически отображают списки атрибутов объектов внутри своих графических пользовательских интерфейсов и могут трактоваться как альтернативы `dir`. Скажем, IDLE будет выводить список атрибутов объекта во всплывающем окне выбора, когда вы наберете точку после имени объекта и сделаете паузу или нажмете клавишу `<Tab>`. Тем не менее, это главным образом считается средством автоматического завершения, а не источником информации. Дополнительные сведения о среде IDLE приводились в главе 3.

Строки документации: `__doc__`

Помимо комментариев `#` в Python поддерживается документация, которая автоматически присоединяется к объектам и сохраняется во время выполнения для инспектирования. Синтаксически такие комментарии записываются как строки в начале файлов модулей, а также операторов функций и классов, перед любым другим исполняемым кодом (перед ними допускается размещать комментарии `#`, включая строки `#!` в стиле Unix). Python автоматически помещает текст таких строк, неформально известных как *строки документации*, в атрибуты `__doc__` соответствующих объектов.

Строки документации, определяемые пользователем

Например, пусть имеется файл `docstrings.py` с показанным ниже содержимым. Строки документации присутствуют в начале файла, а также в начале функции и класса внутри файла. Здесь для многострочных комментариев в файле и функции применяются блочные строки в утроенных кавычках, но подойдут строки любого вида; односторонние фрагменты в одинарных или двойных кавычках наподобие приведенного в классе вполне хороши, но они не разрешают записывать многострочный текст. Мы пока еще не изучали операторы `def` и `class`, поэтому пока проигнорируйте все за исключением строк в их начале:

```
"""
Module documentation
```

```
Words Go Here
"""
spam = 40

def square(x):
    """
        function documentation
        can we have your liver then?
    """
    return x ** 2          # квадрат

class Employee:
    "class documentation"
    pass

print(square(4))
print(square.__doc__)
```

Весь смысл соблюдения такого протокола документации в том, что после импортирования файла ваши комментарии *предохраняются для инспектирования* в атрибутах `__doc__`. Таким образом, для отображения строк документации, ассоциированных с модулем и его объектами, мы просто импортируем файл и выводим его атрибуты `__doc__`, где Python сохранил текст:

```
>>> import docstrings
16

    function documentation
    can we have your liver then?

>>> print(docstrings.__doc__)
Module documentation
Words Go Here

>>> print(docstrings.square.__doc__)
    function documentation
    can we have your liver then?

>>> print(docstrings.Employee.__doc__)
    class documentation
```

Обратите внимание, что выводить строки документации предпочтительнее с помощью `print`, иначе получится одна строка со встроенными символами `\n`.

Строки документации можно также присоединять к *методам* классов (раскрываются в части VI), но поскольку они представляют собой всего лишь операторы `def`, вложенные в операторы `class`, то не являются особым случаем. Чтобы извлечь строку документации метода из класса внутри модуля, понадобится просто указать путь к классу: `модуль.класс.метод.__doc__` (пример со строками документации методов будет предложен в главе 29).

Стандарты и приоритеты строк документации

Как упоминалось ранее, общепринятая практика на сегодняшний день рекомендует использовать комментарии `#` только для ограниченного документирования выражений, операторов или небольших групп операторов. Строки документации лучше применять для документирования функциональности более высокого уровня и охвата в файлах, функциях или классах, к тому же они стали ожидаемой частью программного обеспечения Python. Но кроме таких рекомендаций вам все равно придется решить, что именно писать.

Хотя в определенных компаниях есть внутренние стандарты, не существует какого-то четкого стандарта относительно того, что должно помещаться в текст строк документации. В свое время предлагались разнообразные языки разметки и шаблоны (например, HTML или XML), но они, похоже, не обрели популярность в мире Python. Откровенно говоря, убедить программистов на Python в необходимости документировать свой код, используя вручную написанную HTML-разметку, вряд ли удастся в наше время. Может быть, мы требуем слишком много, и это неприменимо к документированию кода в целом.

Для некоторых программистов документация имеет тенденцию обладать более низким приоритетом, нежели должна. Слишком часто мы считаем себя удачливыми, если обнаруживаем в файле хоть какие-нибудь комментарии (и даже больше, когда они точны и актуальны). Я настоятельно рекомендую вам свободно документировать свой код – документация действительно является важной частью хорошо написанных программ. Однако имейте в виду, что на сегодняшний день не существует стандартов, регламентирующих структуру строк документации; если вы хотите их использовать, то сейчас подходит все, что угодно. Как и в случае написания самого кода, именно от вас зависит, создавать ли документацию и поддерживать ее в актуальном состоянии, но в такой задаче вашим лучшим союзником, вероятно, будет здравый смысл.

Встроенные строки документации

Как выясняется, встроенные модули и объекты в Python применяют похожие методики для присоединения документации помимо списков атрибутов, возвращающихся функцией `dir`. Скажем, чтобы увидеть фактическое описание встроенного модуля, предназначенное для человека, импортируйте модуль и выведите его строку `__doc__`:

```
>>> import sys
>>> print(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
...остальной текст не показан...
```

Функции, классы и методы внутри встроенных модулей также имеют присоединенные описания в своих атрибутах `__doc__`:

```
>>> print(sys.getrefcount.__doc__)
getrefcount(object) -> integer

Return the reference count of object. The count returned is generally
one higher than you might expect, because it includes the (temporary)
reference as an argument to getrefcount().
```

Получить сведения о встроенных функциях можно также через их строки документации:

```
>>> print(int.__doc__)
int(x[, base]) -> integer

Convert a string or number to an integer, if possible. A floating
point argument will be truncated towards zero (this does not include a
...остальной текст не показан...
```

```
>>> print(map.__doc__)
map(func, *iterables) --> map object
Make an iterator that computes the function using arguments from
each of the iterables. Stops when the shortest iterable is exhausted.
```

Вы можете получить массу информации о встроенных инструментах, инспектируя подобным образом их строки документации, но поступать так вовсе необязательно — функция `help`, рассматриваемая в следующем разделе, делает это автоматически.

PyDoc: функция `help`

Методика со строками документации оказалась настолько полезной, что в Python со временем появился инструмент, который еще больше облегчает отображение строк документации. Стандартный инструмент *PyDoc* представляет собой код Python, которому известно, как извлекать строки документации вместе с ассоциированной структурной информацией и форматировать их в аккуратно организованные отчеты различных видов. В области открытого кода доступны дополнительные инструменты для извлечения и форматирования строк документации (в том числе инструменты, способные поддерживать структурированный текст — поищите сведения в веб-сети), но Python поставляется с инструментом PyDoc в своей стандартной библиотеке.

Существует несколько способов запуска PyDoc, включая параметры сценария командной строки, которые могут сохранять результирующую документацию для просмотра в будущем (они описаны далее, а также в руководстве по библиотеке Python). Пожалуй, двумя наиболее заметными интерфейсами PyDoc являются встроенная функция `help`, а также средства отображения отчетов в формате HTML с графическим пользовательским интерфейсом и для веб-сети. Функция `help` кратко упоминалась в главе 4; она вызывает PyDoc с целью генерации простого текстового отчета для любого объекта Python. В таком режиме справочный текст выглядит очень похожим на страницы, выводимые командой `man` в Unix-подобных системах, и фактически разбивает длинный текст на страницы подобно Unix-команде `more` за пределами графических пользовательских интерфейсов вроде IDLE (нажмайте клавишу пробела для перехода на следующую страницу, `<Enter>` для перехода на следующую строку и `<Q>` для выхода):

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount in module sys:
getrefcount(...)
    getrefcount(object) -> integer
    Return the reference count of object. The count returned is generally
    one higher than you might expect, because it includes the (temporary)
    reference as an argument to getrefcount().
```

Обратите внимание, что вы не обязаны импортировать `sys` для вызова `help`, но должны импортировать `sys`, чтобы получить справку по `sys` таким способом; функция `help` ожидает, что ей будет передана ссылка на объект. В Python 3.X и 2.7 можно получить справку по модулю, который не был импортирован, поместив имя модуля в кавычки как строку, например, `help('re')`, `help('email.message')`, но поддержка такого приема и других режимов может отличаться между версиями Python.

Для более крупных объектов, таких как модули и классы, отображаемая функцией `help` информация разбивается на множество разделов, вводная часть которых пока-

зана ниже. Запустите вызов в интерактивной подсказке, чтобы увидеть полный отчет (получен в версии Python 3.7):

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

MODULE REFERENCE
    https://docs.python.org/3.7/library/sys
    ...остальной текст не показан...

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.
    ...остальной текст не показан...

FUNCTIONS
    __breakpointhook__ = breakpointhook(...)
        breakpointhook(*args, **kws)
    ...остальной текст не показан...

DATA
    __stderr__ = None
    __stdin__ = None
    __stdout__ = None
    api_version = 1013
    ...остальной текст не показан...

FILE
    (built-in)
```

Часть информации в отчете – это строки документации, а часть (скажем, шаблоны вызова функций) – структурная информация, которую РуДос автоматически собирает, инспектируя внутренности объектов, когда они доступны.

Кроме модулей вы также можете использовать `help` на встроенных функциях, методах и типах. Применение слегка варьируется между версиями Python, но чтобы получить справку по *встроенному типу*, попробуйте указать имя типа (например, `dict` для словаря, `str` для строки, `list` для списка), фактический объект нужного типа (скажем, `{}`, `''`, `[]`) или метод с фактическим объектом либо именем типа (например, `str.join`, `''.join`)¹. Вы получите крупное визуальное представление с описанием всех методов, доступных для данного типа, либо использования данного метода:

```
>>> help(dict)
Help on class dict in module builtins:

class dict(object)
| dict() -> new empty dictionary.
| dict(mapping) -> new dictionary initialized from a mapping object's
...остальной текст не показан...
```

¹ Имейте в виду, что запросить справку на фактическом *строковом объекте* напрямую (например, `help('')`) в последних версиях Python не удастся: обычно вы не получите никакой помощи, потому что строки интерпретируются специальным образом – скажем, как запрос справки для модуля, который не был импортирован (см. ранее). В таком контексте придется использовать имя типа `str`, хотя другие типы фактических объектов (`help([])`) и имен строковых методов, указываемых через фактические объекты (`help(''.join)`), работают нормально (по крайней мере, в версии Python 3.7 – со временем это может измениться). Доступен также режим интерактивной справки, который запускается набором просто `help()`.

```
>>> help(str.replace)
Help on method_descriptor:
replace(self, old, new, count=-1, /)
    Return a copy with all occurrences of substring old replaced by new.
    ...остальной текст не показан...

>>> help('.replace')
...похоже на предыдущий результат...
```

```
>>> help(ord)
Help on built-in function ord in module builtins:
ord(c, /)
    Return the Unicode code point for a one-character string.
```

Наконец функция `help` работает с вашими модулями настолько же хорошо, как и с встроенными. Ниже показан отчет по созданному ранее файлу `docstrings.py`. И снова часть вывода — это строки документации, а часть — информация, автоматически извлеченная при инспектировании структур объектов:

```
>>> import docstrings
>>> help(docstrings.square)
Help on function square in module docstrings:
square(x)
    function documentation
    can we have your liver then?
>>> help(docstrings.Employee)
Help on class Employee in module docstrings:
class Employee(builtins.object)
|   class documentation
|
...остальной текст не показан...

>>> help(docstrings)
Help on module docstrings:
NAME
    docstrings
DESCRIPTION
    Module documentation
    Words Go Here
CLASSES
    builtins.object
        Employee
        .
    class Employee(builtins.object)
        |   class documentation
        |
...остальной текст не показан...

FUNCTIONS
    square(x)
        function documentation
        can we have your liver then?

DATA
    spam = 40
FILE
    c:\code\docstrings.py
```

PyDoc: отчеты в формате HTML

Текстовых отображений функции `help` достаточно во многих контекстах, особенно в интерактивной подсказке. Тем не менее, читателям, привыкшим к более развитым способам представления информации, они могут показаться несколько примитивными. В настоящем разделе обсуждается разновидность PyDoc, основанная на HTML, которая более наглядно визуализирует документацию по модулям для просмотра в веб-браузере и способна даже открывать ее автоматически. В версии Python 3.3 способ запуска данного инструмента изменился.

- До версии Python 3.3 в состав Python входил простой настольный клиент с графическим пользовательским интерфейсом, предназначенный для отправки поисковых запросов. Этот клиент запускал веб-браузер для просмотра документации, произведенной автоматически стартовавшим локальным сервером.
- Начиная с версии Python 3.3, предшествующий клиент с графическим пользовательским интерфейсом заменен интерфейсной схемой, поддерживающей единый браузер, которая сочетает поиск и отображение на веб-странице, взаимодействующей с автоматически запущенным локальным сервером.
- В версии Python 3.2 несовпадение было устранено за счет поддержки как первоначального клиента с графическим пользовательским интерфейсом, так и более нового режима с единым браузером, ставшего обязательным в выпуске Python 3.3.

Поскольку аудитория настоящей книги включает пользователей последних и наилучших, а также испытанных на практике версий Python, здесь мы исследуем обе схемы. Помните о том, что отличия между двумя схемами имеют отношение лишь к верхнему уровню их пользовательских интерфейсов. Отображаемая ими документация почти идентична и в любом режиме работы PyDoc может также применяться для генерации текста в консоли и HTML-файлов для просмотра предпочтительным способом.

Python 3.2 и последующие версии: режим с единым браузером в PyDoc

Начиная с версии Python 3.3, первоначальный режим клиента с графическим пользовательским интерфейсом PyDoc, представленный в Python 2.X и более ранних выпусках Python 3.X, больше не доступен. Этот режим инициировался в Python 3.2 с помощью элемента меню `Module Docs` (Документация по модулям) в Windows 7 и предшествующих версиях и через команду `pydoc -g` в командной строке. В версии Python 3.2 он был объявлен устаревшим, хотя нужно внимательно присмотреться — на моем компьютере с Python 3.2 он нормально работает, не выдавая никаких предупреждений.

Однако в Python 3.3 данный режим вообще исчез; он был заменен командой `pydoc -b`, которая взамен порождает процессы локально выполняющегося сервера документации, а также веб-браузера, который функционирует как клиент поискового механизма и средство отображения страниц. Браузер изначально открывается на странице индекса модулей с расширенной функциональностью. Существуют дополнительные способы использования PyDoc (скажем, сохранение HTML-страницы в файл для просмотра в более позднее время, как будет описано далее), так что это относительно небольшое рабочее изменение.

Чтобы запустить более новый режим только для браузера PyDoc в Python 3.2 и последующих версиях, достаточно перечисленных ниже команд: ради удобства все они применяют аргумент `-m` командной строки Python для нахождения файла модуля PyDoc в пути поиска импортируемых модулей. В первой команде предполагается, что каталог с Python присутствует в системной переменной среды PATH; во второй ко-

манде задействован новый запускающий модуль Windows, появившийся в Python 3.3; в третьей команде указан полный путь к каталогу с Python, если другие две схемы не работают. В приложении А второго тома приведены добавочные сведения об аргументе `-m`, а в приложении Б второго тома – о запускающем модуле Windows.

```
c:\code> python -m pydoc -b
Server ready at http://localhost:62135/
Server commands: [b]rowser, [q]uit
server> q
Server stopped

c:\code> py -3 -m pydoc -b
Server ready at http://localhost:62144/
Server commands: [b]rowser, [q]uit
server> q
Server stopped

c:\code> C:\python33\python -m pydoc -b
Server ready at http://localhost:62153/
Server commands: [b]rowser, [q]uit
server> q
Server stopped
```

Результатом ввода одной из таких команд будет запуск PyDoc как локально выполняющегося веб-сервера на выделенном (но по умолчанию произвольно выбираемом из числа неиспользуемых) порте и открытию веб-браузера, который действует в качестве клиента и отображает страницу со ссылками на документацию для всех модулей, импортируемых в пути поиска модулей (включая каталог, где запущен PyDoc). Интерфейс веб-страницы верхнего уровня PyDoc представлен на рис. 15.1.



Рис. 15.1. Стартовая индексная страница верхнего уровня HTML-интерфейса PyDoc с единым браузером в Python 3.2 и последующих версиях, который стал заменой клиента с графическим пользовательским интерфейсом из более ранних версий Python, начиная с Python 3.3

Помимо индекса модулей веб-страница PyDoc в своей верхней части также содержит поля ввода для запрашивания страницы документации по индивидуальному модулю (слева от кнопки `Get` (Получить)) и поиска связанных записей (слева от кнопки `Search` (Искать)), которые заменяют поля в прежнем клиенте с графическим пользовательским интерфейсом. Здесь также можно щелкать на ссылках `Module Index` (Индекс модулей) для перехода на стартовую страницу, `Topics` (Темы) для перехода на общие темы Python и `Keywords` (Ключевые слова) для перехода к обзорам операторов и ряда выражений.

Следует отметить, что на индексной странице будут перечисляться и *модули*, и *сценарии* верхнего уровня в текущем каталоге – `C:\code` с примерами для книги, где инструмент PyDoc запускается рассмотренными выше командами. Хотя PyDoc главным образом предназначен для отображения документации по импортируемым модулям, его иногда можно применять для показа документации по сценариям. Выбранный файл должен быть импортирован, чтобы его документация визуализировалась, и как вы уже знаете, импортирование приводит к выполнению кода в файле. Модули при выполнении по обыкновению лишь определяют инструменты, так что обычно это не имеет значения.

Тем не менее, если вы запрашиваете документацию для файла сценария верхнего уровня, то окно командной оболочки, где запускался инструмент PyDoc, служит стандартным вводом и выводом для любого взаимодействия с пользователем. Совокупный эффект в том, что страница документации по сценарию будет появляться после того, как он запустится, и после отображения его вывода в окне командной оболочки. Однако для одних сценариев иногда это работает лучше, чем для других; скажем, интерактивный ввод может странно чередоваться с собственными приглашениями к вводу команд сервера PyDoc.

После того как вы покинете новую стартовую страницу на рис. 15.1, страницы документации по индивидуальным модулям окажутся одинаковыми в более новом режиме с единым браузером и начальном клиенте с графическим пользовательским интерфейсом, не считая дополнительных полей ввода в верхней части страницы в первом случае. Например, на рис. 15.2 показаны страницы нового отображения документации – открытые для двух определяемых пользователем модулях, которые мы напишем в следующей части книги в рамках учебного примера оценочных испытаний в главе 21. В любой из двух схем страницы документации содержат автоматически созданные гиперссылки, которые позволяют переходить на документацию по связанным компонентам внутри приложения. Скажем, вы обнаружите ссылки также для открытия страниц импортированных модулей.

По причине сходства страниц отображения материал следующего раздела о PyDoc до Python 3.2 и приведенные в нем экранные снимки почти полностью применимы также и к Python 3.2, поэтому обязательно ознакомьтесь с дополнительными замечаниями, даже если пользуетесь более новой версией Python. В действительности PyDoc в Python 3.3 и последующих версиях просто устраняет “посредника” в форме клиента с графическим пользовательским интерфейсом, который имеется в версиях до Python 3.2, одновременно оставляя его браузер и сервер.

PyDoc в Python 3.3 по-прежнему поддерживает остальные режимы использования, которые были доступны ранее. Например, команда `pydoc -p` порт устанавливает порт, на котором запускается сервер PyDoc, а команда `pydoc -w` модуль записывает HTML-документацию по модулю в файл с именем `модуль.html` для просмотра в будущем. Удален только режим клиента с графическим пользовательским интерфейсом `pydoc -g` и заменен режимом `pydoc -b`.



Рис. 15.2. Страница документации PyDoc для модулей в Python 3.2 и последующих версиях с полями ввода в верхней части, отображающая информацию о модулях, которые мы напишем в следующей части книги (в главе 21)

Можно также запускать PyDoc для генерации документации в виде простого текста (вариант, похожий на вывод команды `man` в Unix-подобных системах, упоминаемый ранее в главе) — следующая командная строка эквивалентна вызову `help` в интерактивной подсказке Python:

```
c:\code> py -3 -m pydoc timeit # Текстовая справка в командной строке
c:\code> py -3
>>> help("timeit") # Текстовая справка в интерактивной подсказке
```

Как интерактивную систему веб-интерфейс PyDoc лучше всего применять для опробования, поэтому здесь мы лишь кратко останавливаемся на деталях его использования; за дополнительными сведениями и параметрами командной строки обращайтесь в руководства Python. Также обратите внимание, что функциональность сервера и браузера PyDoc поступает практически “даром” от инструментов, которые автоматизируют ее в переносимых модулях стандартной библиотеки Python (например, `webbrowser`, `http.server`). Детали и идеи можно почерпнуть из Python-кода PyDoc, находящегося в файле `pydoc.py` стандартной библиотеки.

Изменение цветов PyDoc

После запуска инструмента PyDoc может выясниться, что выбранные в нем цвета по каким-то причинам вам не нравятся. К сожалению, в настоящее время нет простого способа настройки цветов PyDoc. Они жестко заданы глубоко в исходном коде; их нельзя передавать в аргументах функциям или командным строкам либо изменять в конфигурационных файлах или глобальных переменных в самом модуле.

Но в системе с открытым кодом вы всегда можете изменить код — PyDoc находится в файле `pydoc.py`, входящем в состав стандартной библиотеки Python, который в системе Windows расположен в каталоге `C:\Python\lib`. Повсюду в коде цвета представлены в виде шестнадцатеричных строк со значениями RGB. Скажем, строка `'#eeaa77'` определяет 3-байтовое (24-битное) значение с 2 шестнадцатеричными цифрами, указывающими 1-байтовые (8-битные) красную, зеленую и синью цветовые составляющие (десятичные 238, 170 и 119), что в результате дает оттенок оранжевого для заголовков функций. Стока `'#ee77aa'` визуализируется в темный розовый цвет, применяемый в девяти местах, включая заголовки классов и индексной страницы.

В целях настройки найдите такие строки со значениями цвета и замените их предпочтительными вариантами. В среде IDLE для этого понадобится выбрать в меню `Edit` (Правка) пункт `Find` (Найти) и указать регулярное выражение `#\w{6}` (согласно синтаксису сопоставления с образцом модуля `re` в Python оно соответствует шести алфавитно-цифровым символам после `#`; за деталями обращайтесь в руководство по библиотеке).

Для подбора цветов большинство программ с диалоговыми окнами выбора цвета позволяют получать значения RGB; в состав примеров входит сценарий с графическим пользовательским интерфейсом `setcolor.py`, который делает то же самое. В своей копии PyDoc я заменил все значения `#ee77aa` значением `#008080` (зеленовато-голубой), чтобы избавиться от темного розового цвета. Замена `#ffc8d8` значением `#c0c0c0` (серый) аналогично меняет светлый розовый цвет фона, на котором отображаются строки документации классов. Такая работа явно не для слабонервных (файл PyDoc содержит выше 2 600 строк), но будет хорошим упражнением по сопровождению кода. Будьте осторожны при замене цветов вроде `ffffff` и `#000000` (белый и черный) и обязательно заранее сделайте резервную копию `pydoc.py`, чтобы была возможность восстановления. В файле `pydoc.py` используются инструменты, которые нам еще не встречались, но при внесении тактических изменений подобного рода вы можете спокойно игнорировать остаток кода.

Обязательно следите за изменениями PyDoc в плане конфигурации, что выглядит главным кандидатом на улучшение. Фактически усилия в данном направлении уже предпринимаются: задача 10716 в списке разработчиков Python касается изменения PyDoc для поддержки таблиц стилей CSS, что усовершенствует возможности настройки со стороны пользователей. В случае успешного решения пользователи могут получить в свое распоряжение средства изменения цветов и других аспектов отображения во внешних CSS-файлах, а не в исходном коде PyDoc.

С другой стороны, в текущий момент это запланировано лишь в версии Python 3.8 и будет требовать от пользователей PyDoc также знания кода CSS. К сожалению, он сам по себе обладает нетривиальной структурой, которую многие программисты на Python могут понимать не настолько хорошо, чтобы вносить изменения. На момент написания главы предлагаемый CSS-файл PyDoc уже содержал 227 строк кода, которые вероятно мало что значат для людей, не знакомых с веб-разработкой (и вряд ли разумно предлагать им изучить какой-то инструмент веб-разработки, просто чтобы подстроить PyDoc!).

Начиная с версии Python 3.3, в PyDoc поддерживается таблица стилей CSS, которая предоставляет ряд вариантов настройки, но без особого энтузиазма, и поставляется пустой. До тех пор, пока ситуация не прояснится, внесение изменений в код PyDoc выглядит лучшим вариантом. Так или иначе, таблицы стилей CSS выходят далеко за рамки тематики настоящей книги — подробности ищите в веб-сети и просматривайте пояснительные записи к будущим выпускам Python на предмет разработки PyDoc.

Python 3.2 и предшествующие версии: клиент с графическим пользовательским интерфейсом

В этом разделе будет описан первоначальный режим клиента с графическим пользовательским интерфейсом PyDoc для тех читателей, кто работает с Python 3.2 и предшествующими версиями, а также дополняется контекст PyDoc в целом. Он основан на материалах предыдущего раздела, которые здесь не повторяются, так что если вы применяете более старую версию Python, то хотя бы бегло просмотрите предыдущий раздел.

Как упоминалось ранее, инструмент PyDoc в версии Python 3.2 предоставлял графический пользовательский интерфейс верхнего уровня — простой, но переносимый сценарий Python/tkinter для отправки запросов — плюс сервер документации. Запросы из клиента направлялись серверу, который производил отчеты, отображаемые во всплывающем окне веб-браузера. Не считая необходимости отправки запросов, процесс является почти полностью автоматическим.

Чтобы инициировать PyDoc в таком режиме, обычно сначала нужно запустить графический пользовательский интерфейс поискового механизма (рис. 15.3). Стартовать его можно либо выбором элемента меню *Module Docs* в Windows 7 и предшествующих версиях, либо запуском сценария *pydoc.py* с аргументом командной строки *-g* из каталога стандартной библиотеки Python: в Windows это *Lib*, но можно также указать флаг *-m* для Python, чтобы избежать набора пути к сценарию:

```
c:\code> c:\python32\python -m pydoc -g      # Явный путь к Python  
c:\code> py -3.2 -m pydoc -g                  # Версия с запускающим модулем  
                                                # Windows в Python 3.3+
```

Введите имя интересующего модуля и нажмите клавишу <Enter>; PyDoc пройдет по пути поиска импортируемых модулей (*sys.path*) в попытке найти запрошенный модуль и сослаться на него.

Отыскав необходимую запись, выберите ее и щелкните на кнопке *go to selected* (перейти к выбранному). PyDoc породит процесс веб-браузера для отображения отчета, визуализированного в формате HTML. На рис. 15.4 показано, как PyDoc отображает информацию для встроенного модуля *glob*. Обратите внимание на гиперссылки в разделе *Modules* (Модули) данной страницы — вы можете щелкнуть на них и переходить на страницы PyDoc для связанных (импортированных) модулей. В случае крупной страницы PyDoc также генерирует гиперссылки на разделы внутри страницы.

Рис. 15.3. Клиент с графическим пользовательским интерфейсом поискового механизма PyDoc верхнего уровня в Python 3.2 и предшествующих версиях: введите имя модуля, для которого хотите получить документацию, нажмите <Enter>, выберите модуль и щелкните на кнопке go to selected (перейти к выбранному) или щелкните на кнопке open browser (открыть браузер), не вводя имя модуля, чтобы увидеть все доступные модули

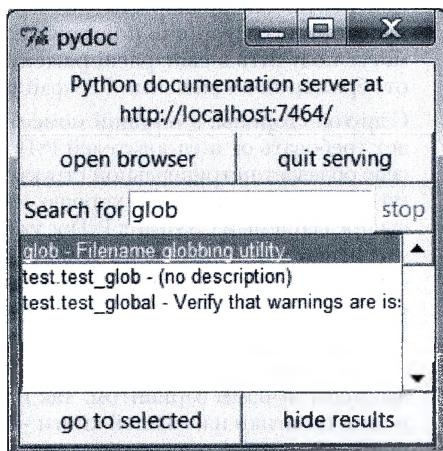




Рис. 15.4. После того, как вы нашли модуль в графическом пользовательском интерфейсе, показанном на рис. 15.3 (вроде стандартного библиотечного модуля) и щелкнули на кнопке go to selected, документация визуализируется в формате HTML и отображается в окне веб-браузера

Подобно интерфейсу функции `help` графический пользовательский интерфейс работает как со встроенными модулями, так и с модулями, определяемыми пользователем. На рис. 15.5 представлена страница, сгенерированная для ранее написанного модуля `docstrings.py`.

Удостоверьтесь, что каталог, в котором хранится файл вашего модуля, содержится в пути поиска импортируемых модулей — как упоминалось, PyDoc должен быть в состоянии импортировать файл, чтобы визуализировать его документацию. Это касается также текущего рабочего каталога — PyDoc может не проверять каталог, из которого он был запущен (что в любом случае не имеет смысла при запуске через элемент меню `Module Docs`), поэтому вам понадобится соответствующим образом расширить переменную среды `PYTHONPATH`. В Python 3.2 и 2.7 необходимо добавить `.` в `PYTHONPATH`, чтобы заставить клиент с графическим пользовательским интерфейсом PyDoc просматривать каталог, из которого он был запущен в командной строке:

```
c:\code> set PYTHONPATH=.;%PYTHONPATH%
c:\code> py -3.2 -m pydoc -g
```

Такая установка также требовалась для просмотра текущего каталога в новом режиме с единым браузером `pydoc -b` в Python 3.2. Тем не менее, в Python 3.3 и последующих версиях `.` включается автоматически, так что никакие установки путей не нужны — незначительное, но заслуживающее внимания улучшение.

PyDoc можно настраивать и запускать разнообразными способами, которые здесь рассматриваться не будут; за дополнительными сведениями обращайтесь в руководство по стандартной библиотеке Python.



Рис. 15.5. Инструмент PyDoc способен обслуживать страницы документации для встроенных и определяемых пользователем модулей в пути поиска импортируемых модулей. Здесь страница для модуля, определяемого пользователем, отображает все его строки документации, извлеченные из файла исходного кода

Главное, что следует вынести из этого раздела – по существу PyDoc предлагает реализацию отчетов “даром”; если вы эффективно используете строки документации в своих файлах, то PyDoc выполнит всю работу по их сбору и форматированию с целью отображения. PyDoc выдает справку только для таких объектов, как функции и модули, но предоставляет легкий способ доступа к среднему уровню документации по инструментам подобного рода – его отчеты более полезны, чем обычные списки атрибутов, и менее всесторонни, нежели стандартные руководства.

PyDoc можно также запускать для сохранения документации в формате HTML по модулю в файле, чтобы позже просмотреть или распечатать; указания были даны в предыдущем разделе. Кроме того, имейте в виду, что PyDoc может не очень хорошо работать при запуске на *сценариях*, которые производят чтение из стандартного ввода – PyDoc импортирует целевой модуль для инспектирования его содержимого, и когда он выполняется в режиме с графическим пользовательским интерфейсом, может отсутствовать связь с текстом из стандартного ввода, особенно в случае запуска посредством элемента меню Module Docs. Однако модули, которые могут быть импортированы без требований непосредственного ввода, всегда будут нормально обслуживаться PyDoc. Вспомните также замечания относительно сценариев в режиме *-b* инструмента PyDoc в Python 3.2 и последующих версиях, приведенные в предыдущем разделе; запуск PyDoc в режиме с графическим пользовательским интерфейсом из командной строки работает аналогично – вы взаимодействуете в окне запуска.



Трюк дня в клиенте с графическим пользовательским интерфейсом PyDoc. Если вы щелкнете на кнопке open browser в окне, изображенном на рис. 15.3, тогда PyDoc произведет индексную страницу, содержащую гиперссылку на каждый модуль, который вы в состоянии импортировать на своем компьютере. Сюда входят стандартные библиотечные модули Python, модули установленных сторонних расширений, определяемые пользователем модули в пути поиска импортируемых модулей и даже статически или динамически компонуемые модули, написанные на С. Извлечь такую информацию по-другому нелегко без написания кода, который инспектирует все источники модулей. В Python 3.2 это иногда требуется делать сразу после открытия пользовательского интерфейса, поскольку он может не работать полноценно по завершении поисков. Также обратите внимание, что в интерфейсе – в с единым браузером PyDoc в Python 3.2 и последующих версиях вы получаете ту же самую функциональность индекса на его старовой странице верхнего уровня, которая была показана на рис. 15.1.

За рамками строк документации: Sphinx

Если вы ищете более передовой способ документирования своей системы Python, тогда можете опробовать *Sphinx* (<http://sphinx-doc.org>). Инструмент Sphinx применяется стандартной документацией Python, описанной в следующем разделе, и многочисленными другими проектами. Он использует простой язык разметки *reStructuredText* и многое наследует из комплекта *Docutils* инструментов разбора и трансляции *reStructuredText*.

Помимо прочего Sphinx поддерживает разнообразные выходные форматы (HTML, включая Windows HTML Help, LaTeX для печатаемых версий PDF, страницы руководства и простой текст); всесторонние и автоматические перекрестные ссылки; иерархическую структуру с автоматическими ссылками на связанные материалы; автоматические индексы; автоматическую цветовую раскраску синтаксиса с применением *Pygments* (сам по себе известный инструмент Python); и многое другое. Вероятно использование Sphinx для небольших программ, где вполне достаточно строк документации и PyDoc, будет излишеством, но в случае крупных проектов Sphinx способен выдавать документацию профессионального уровня. Дополнительные сведения о Sphinx и связанных инструментах ищите на указанном веб-сайте.

Стандартный набор руководств

Стандартные руководства Python служат полным и наиболее актуальным описанием языка и его инструментального набора. Руководства Python поставляются в формате HTML и других форматах. Они устанавливаются с системой Python в среде Windows и доступны через меню кнопки Пуск, а также могут открываться из меню Help (Справка) внутри IDLE. Вы также можете получить набор руководств на веб-сайте <http://www.python.org> в различных форматах или читать его прямо там в онлайновом режиме (проследовав по ссылке Documentation (Документация)). В Windows руководства организованы как скомпилированный справочный файл с поддержкой поиска, а онлайновые версии на веб-сайте Python включают страницу веб-поиска.

После открытия руководства в формате Windows отображают корневую страницу, подобную приведенной на рис. 15.6, показывая содержимое локальной копии в системе Windows. Двумя самыми важными разделами, вероятно, следует считать The Python Standard Library (Стандартная библиотека Python), где документируются встроенные типы, функции, исключения и стандартные библиотечные модули, а также The Python

Language Reference (Справочник по языку Python), который предоставляет формальное описание деталей языкового уровня. В разделе The Python Tutorial (Учебное пособие по Python) предлагается краткое введение для новичков, за рамки которого вы, пожалуй, уже вышли.

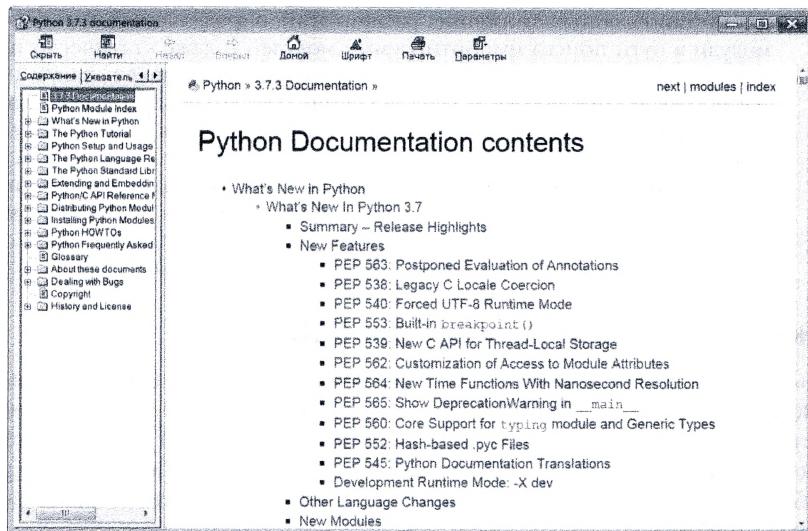


Рис. 15.6. Стандартный набор руководств Python, доступный в онлайновом режиме на веб-сайте <http://www.python.org>, посредством меню Help в IDLE и через меню кнопки Пуск в Windows 7 и предшествующих версиях. В Windows он представляет собой справочный файл с возможностью поиска и есть также поисковый механизм для онлайновой версии. Чаще всего придется использовать раздел The Python Standard Library

Интересно отметить, что в разделе What's New in Python (Что нового в Python) стандартного набора руководств ведется хроника изменений, внесенных в каждый выпуск Python, начиная с версии Python 2.0, которая появилась в конце 2000 года. Раздел What's New in Python полезен для тех, кто занимается переносом старого кода Python или обладает опытом работы в старых версиях Python. Такие документы особенно пригодны для выявления дополнительных деталей об отличиях в линейках языка Python 2.X и 3.X, рассматриваемых в данной книге, а также в их стандартных библиотеках.

Веб-ресурсы

На официальном веб-сайте Python (<http://www.python.org>) вы найдете ссылки на разнообразные ресурсы Python, где раскрываются специальные темы или предметные области. Щелкните на ссылке Documentation для доступа к онлайновому руководству и учебнику для начинающих в Python (Beginner's Guide to Python). На веб-сайте также доступны неанглоязычные ресурсы по Python и введения, ориентированные на разные целевые аудитории.

В наши дни вы также обнаружите многочисленные вики-страницы, блоги, веб-сайты и массу других ресурсов, посвященных Python. Чтобы обратиться к онлайновому сообществу, поищите в Google термин вроде “Python programming” (“программирование на Python”) или на любую интересующую тему; вы наверняка найдете изобилие материала для просмотра.

Изданные книги

В качестве финального ресурса вы можете выбрать из коллекции прошедших профессиональную редактуру и опубликованных справочников для Python. Имейте в виду, что книги имеют тенденцию отставать от самых современных изменений Python, отчасти из-за объема работы, связанной с написанием, и частично по причине естественных задержек, присущих издательскому циклу. Как правило, к моменту выхода книга отстает на три и больше месяцев от текущего состояния Python (проверьте мне – мои книги обладают дурной привычкой незначительно устаревать от момента их написания ко времени их попадания на полки!). В отличие от стандартных руководств книги также обычно не достаются бесплатно.

Тем не менее, для многих удобство и качество изданных книг стоит потраченных денег. Кроме того, Python изменяется настолько медленно, что книги остаются актуальными на протяжении многих лет после их выхода в свет, особенно если авторы публикуют изменения в веб-сети. В предисловии приведены ссылки на другие книги по Python.

Распространенные затруднения при написании кода

Перед упражнениями по программированию для текущей части книги давайте пройдемся по ряду самых распространенных ошибок, допускаемых начинающими при написании кода операторов и программ на Python. Многие из них являются предупреждениями, которые приводились ранее в данной части, а здесь они для удобства собраны в одном месте. Набравшись опыта программирования на Python, вы научитесь избегать таких ловушек, но несколько слов в настоящий момент могут помочь изначально не попадать в некоторые из них.

- **Не забывайте о двоеточиях.** Постоянно помните о необходимости набора символа : в конце заголовка составного оператора – первой строки if, while, for и т.д. Вероятно, поначалу вы будете забывать об этом (как было со мной и большинством из моих почти 4000 студентов, которых я обучал Python на протяжении многих лет), но можете утешиться тем фактом, что вскоре у вас выработается неосознанная привычка.
- **Начинайте в колонке 1.** Удостоверьтесь, что начинаете код верхнего уровня (невложенный) в колонке 1. Это касается невложенного кода, вводимого в файлах модулей, равно как и кода, набираемого в интерактивной подсказке.
- **Пустые строки значимы в интерактивной подсказке.** Пустые строки в составных операторах, размещенных внутри файлов модулей, всегда несущественны и игнорируются, но при наборе кода в интерактивной подсказке они заканчивают операторы. Другими словами, пустая строка сообщает интерактивной командной оболочке о том, что составной оператор завершен; если необходимо продолжить, тогда не следует нажимать клавишу <Enter>, видя приглашение ... (или в IDLE), пока набор составного оператора действительно не будет закончен. Вдобавок это также означает невозможность вставки многострочного кода в интерактивной подсказке; нужно запускать по одному полному оператору за раз.
- **Делайте отступы согласованно.** Избегайте смешивания табуляций и пробелов в отступах блока, если только не знаете, что ваш текстовый редактор делает с табуляциями. В противном случае то, что вы видите в своем редакторе, может быть не тем, что видит Python, когда он пересчитывает табуляции в пробелы.

Сказанное справедливо не только для Python, но для любого блочно-структурированного языка — если у следующего программиста табуляции настроены по-другому, то ему будет сложно или невозможно понять структуру вашего кода. Надежнее использовать для каждого блока все табуляции или все пробелы.

- **Не пишите код Python в стиле C.** Напоминание программистам на C/C++: нет необходимости помещать в круглые скобки выражения проверок в заголовках `if` и `while` (например, `if (X==1) :`). При желании поступать так можно (заключать в круглые скобки разрешено любое выражение), но в данном контексте они совершенно излишни. Кроме того, не заканчивайте все свои операторы точками с запятой; формально делать это в Python законно, но абсолютно бесполезно, если только вы не размещаете в одиночной строке более одного оператора (обычно оператор завершается по достижении конца строки). Также запомните: не встраивайте операторы присваивания в проверки циклов `while` и не применяйте `{}` вокруг блоков (взамен согласованно смещайте вложенные блоки кода).
- **Используйте простые циклы `for` вместо `while` или `range`.** Еще одно напоминание: простой цикл `for` (скажем, `for x in seq:`) почти всегда проще в написании и часто быстрее в выполнении, чем цикл с подсчетом на основе `while` или `range`. Поскольку Python внутренне поддерживает индексацию для простого цикла `for`, иногда он может быть быстрее эквивалентного `while`, хотя ситуация зависит от кода и версии Python. Однако исключительно ради простоты кода не поддавайтесь искушению подсчитывать что-либо в Python!
- **Будьте осторожны с изменяемыми объектами в присваиваниях.** Об этом упоминалось в главе 11: нужно проявлять осмотрительность при использовании изменяемых объектов в групповом присваивании (`a = b = []`), а также в дополненном присваивании (`a += [1, 2]`). В обоих случаях изменения на месте могут повлиять на другие переменные. Если вдруг вы подзабыли, почему сказанное справедливо, тогда снова почитайте главу 11.
- **Не ожидайте получить результаты от функций, которые изменяют объекты на месте.** Мы уже сталкивались с этим ранее: операции изменения на месте вроде представленных в главе 8 методов `list.append` и `list.sort` не возвращают значения (за исключением `None`), так что они должны вызываться без присваивания результата. Начинающие нередко применяют оператор наподобие `mylist = mylist.append(X)` в попытке получить результат метода `append`, но на самом деле он присваивает переменной `mylist` объект `None`, а не модифицированный список (фактически будет утрачена ссылка на целый список).
Более коварный пример такого рода относится к коду Python 2.X, в котором предпринимается попытка прохода по элементам словаря в сортированной манере. Довольно часто можно встретить код вида `for k in D.keys().sort():`. Код почти работает — метод `keys` строит список ключей, а метод `sort` упорядочивает его, — но из-за того, что метод `sort` возвращает `None`, цикл терпит неудачу, поскольку в итоге пытается пройти по `None` (не последовательность). В Python 3.X отказ произойдет даже раньше, т.к. ключи словаря являются представлениями, а не списками! Чтобы написать корректный код, необходимо либо использовать более новую встроенную функцию `sorted`, которая возвращает отсортированный список, либо разнести вызовы методов по разным операторам: `Ks = list(D.keys()),` затем `Ks.sort()` и наконец `for k in Ks:`. Кстати, это один из случаев, когда может понадобиться вызывать метод `keys` явно для организации цикла, не полагаясь на словарные итераторы — итераторы не сортируют.

- **Всегда применяйте круглые скобки для вызова функции.** Чтобы вызвать функцию, после ее имени потребуется указать круглые скобки независимо от того, принимает она аргументы или нет (например, используйте `function()`, а не `function`). В следующей части книги будет показано, что функции представляют собой просто объекты, имеющие специальную операцию — вызов, который инициируется с помощью круглых скобок. На них можно ссылаться как на любой другой объект, не делая вызова. Во время обучения такая проблема чаще всего возникает с файлами; чтобы закрыть файл, новички нередко пишут `file.close` вместо `file.close()`. Поскольку вполне законно ссылаться на функцию, не вызывая ее, первая версия без скобок молча выполнится, но не закроет файл!
- **Не используйте расширения или пути при импортировании и перезагрузке.** Опускайте пути по каталогам и файловые расширения в операторах `import` — применяйте `import mod`, а не `import mod.py`. Мы обсуждали основы модулей в главе 3 и продолжим изучение модулей в части V. Из-за того, что модули могут иметь и другие расширения помимо `.py` (скажем, `.russ`), жесткое кодирование специфического расширения не только незаконно с точки зрения синтаксиса, но даже не имеет смысла. Python выбирает расширение автоматически, и любой зависящий от платформы синтаксис путей по каталогам поступает из настроек пути поиска модулей, а не оператора `import`.
- **Прочие затруднения в других частях.** Обязательно просмотрите предостережения, касающиеся встроенных типов, в конце предыдущей части, т.к. их тоже можно квалифицировать в качестве проблем при написании кода. Существуют дополнительные “затруднения”, которые обычно обнаруживаются при написании кода на Python — утрата встроенной функции за счет переназначения ее имени, сокрытие библиотечного модуля из-за использования его имени для одного из своих модулей, модификация стандартных значений изменяемых аргументов и т.д., — но для их раскрытия знаний у вас пока недостаточно. Чтобы узнать больше о том, что должно и что не должно делаться в Python, вы должны продолжить чтение книги; в последующих частях приведенный здесь набор “затруднений” и исправлений будет расширен.

Резюме

В главе был предложен тур по документированию программ, охватывающий как документацию, которую мы пишем сами для собственных программ, так и документацию, доступную для используемых инструментов. Вы ознакомились со строками документации, исследовали онлайновые и автономные ресурсы справочников по Python, а также узнали, каким образом функция `help` из PyDoc и интерфейсы с веб-страницами предоставляют дополнительные источники документации. Поскольку это последняя глава в данной части, был приведен обзор распространенных затруднений при написании кода, который поможет избегать их на практике.

В следующей части книги мы начнем применять то, что уже известно, к более крупным программным конструкциям. В частности, в следующей части раскрывается тема *функций* — инструмента, предназначенного для группирования операторов с целью многократного использования. Но сначала проработайте набор упражнений для текущей части книги, приведенный в конце главы. И прежде чем двигаться дальше, закрепите пройденный материал этой главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Когда должны использоваться строки документации вместо комментариев `#`?
2. Назовите три способа, которыми можно просматривать строки документации.
3. Как можно получить список доступных атрибутов в объекте?
4. Как можно получить список доступных модулей на компьютере?
5. Какую книгу, посвященную Python, вы должны приобрести после этой?

Проверьте свои знания: ответы

1. Строки документации считаются лучшими для крупной документации по функциональности, которая описывает использование модулей, функций, классов и методов в коде. Применение комментариев `#` в наши дни лучше сузить до менее масштабной документации о загадочных выражениях или операторах в стратегически важных местах кода. Отчасти это объясняется тем, что строки документации легче найти в файле исходного кода, но также и тем, что они могут извлекаться и отображаться системой PyDoc.
2. Увидеть строки документации можно с помощью вывода атрибута `_doc_` объекта, путем передачи объекта функции `help` из PyDoc и за счет выбора модулей в основанном на HTML пользовательском интерфейсе PyDoc – либо в режиме `-g` клиента с графическим пользовательским интерфейсом в Python 3.2 и предшествующих версиях, либо в режиме `-b` единого браузера в Python 3.2 и последующих версиях (обязательном, начиная с Python 3.3). Оба запускают клиент-серверную систему, которая отображает документацию во всплывающем окне веб-браузера. PyDoc можно также запустить и сохранить документацию по модулю в HTML-файле с целью просмотра или распечатки в более позднее время.
3. Встроенная функция `dir(X)` возвращает список всех атрибутов, присоединенных к любому объекту. Списковое включение вида `[a for a in dir(X) if not a.startswith('__')]` можно использовать для того, чтобы отфильтровать внутренние имена с подчеркиваниями (в следующей части книги будет показано, как поместить его внутрь функции, облегчив многократное применение).
4. В Python 3.2 и предшествующих версиях можно запустить клиент с графическим пользовательским интерфейсом PyDoc и щелкнуть на кнопке `open browser`, что приведет к открытию веб-страницы, которая содержит ссылки на все модули, доступные вашим программам. Начиная с версии Python 3.3, такой режим с графическим пользовательским интерфейсом больше не работает. В Python 3.2 и последующих версиях вы получаете ту же самую функциональность, запуская более новый режим единого браузера PyDoc посредством ключа командной строки `-b`; стартовая страница верхнего уровня, отображаемая внутри веб-браузера в этом новом режиме, содержит аналогичный список всех доступных модулей.
5. Конечно же, мою. (А если серьезно, то на сегодняшний день книг сотни; в предисловии предложен перечень с несколькими рекомендуемыми книгами, представляющими собой справочники и учебные руководства, и вы должны поискать книги, которые удовлетворят имеющимся потребностям.)

Проверьте свои знания: упражнения для части III

Теперь, когда вы знаете, как пишется базовая программная логика, в следующих упражнениях предлагается выполнить простые задачи с помощью операторов. Самый большой объем работы требует упражнение 4, которое позволит исследовать альтернативные варианты написания кода. Всегда есть много способов приведения в порядок операторов, и частью изучения Python является выяснение, какая организация операторов работает лучше остальных. В конечном итоге вы будете естественным образом стремиться к тому, что опытные программисты на Python называют “установившейся практикой”, но это требует вырабатывания навыков. Решения упражнений находятся в приложении.

1. *Написание базовых циклов.* В данном упражнении предлагается поэкспериментировать с циклами `for`.

- а) Напишите цикл `for`, который выводит код ASCII каждого символа в строке по имени `S`. Для преобразования символа в целочисленный код ASCII используйте встроенную функцию `ord(символ)`. Формально функция `ord` в Python 3.X возвращает кодовую точку Unicode, но если вы ограничите содержимое строки символами ASCII, то будете получать обратно коды ASCII. (Опробуйте ее интерактивно, чтобы посмотреть, как она работает.)
- б) Далее измените цикл для вычисления суммы кодов ASCII всех символов в строке.
- в) Наконец снова модифицируйте код, чтобы возвращать новый список, который *содержит* коды ASCII всех символов в строке. Дает ли выражение `map(ord, S)` похожий результат? Как насчет `[ord(c) for c in S]`? Почему? (Подсказка: см. главу 14.)

2. *Символы обратной косой черты.* Что произойдет при интерактивном наборе следующего кода?

```
for i in range(50):
    print('hello %d\n\'a' % i)
```

Имейте в виду, что при выполнении за пределами интерфейса IDLE этот код может выдать звуковой сигнал, так что вероятно его не стоит запускать в людном помещении! Взамен выдачи звукового сигнала IDLE выводит странные символы, испортив всю шутку (см. управляющие последовательности в табл. 7.2).

3. *Сортировка словарей.* В главе 8 мы выяснили, что словари являются неупорядоченными последовательностями. Напишите цикл `for`, который выводит элементы словаря в отсортированном порядке (по возрастанию). (Подсказка: применяйте словарные методы `keys` и `sort` или более новую встроенную функцию `sorted`.)
4. *Альтернативные варианты программной логики.* Взгляните на показанный ниже код, в котором используется цикл `while` и флаг `found` для поиска в списке степеней 2 значения 2, возведенного в пятую степень (32). Он хранится в файле модуля по имени `power.py`.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5
found = False
i = 0
while not found and i < len(L):
    if 2 ** X == L[i]:
        found = True
    else:
        i = i+1
```

```
if found:
    print('at index', i)
else:
    print(X, 'not found')
C:\book\tests> python power.py
at index 5
```

В том виде, как есть, пример совершенно не следует обычным методикам написания кода Python. Выполните приведенные далее шаги, чтобы улучшить его (при внесении всех изменений вы можете либо набирать код интерактивно, либо сохранять его в файле сценария, запускаемом в командной строке системы — применение файла гораздо упрощает выполнение упражнения).

- a) Сначала перепишите код с конструкцией `else` цикла `while`, чтобы избавиться от флага `found` и финального оператора `if`.
- б) Затем перепишите код для использования цикла `for` с конструкцией `else`, чтобы избавиться от явной логики индексации списка. (Подсказка: для получения индекса элемента применяйте списковый метод `index` — `L.index(X)` возвращает смещение первого элемента `X` в списке `L`.)
- в) Далее полностью устранитесь цикл, переписав код с использованием простого выражения с операцией членства `in`. (За дополнительными сведениями обращайтесь в главу 8 или наберите для тестирования `2 in [1, 2, 3]`.)
- г) Наконец примените цикл `for` и списковый метод `append` для генерации списка степеней 2 (`L`) вместо жесткого кодирования спискового литерала.

Ниже приведены более глубокие рассуждения.

- д) Как вы думаете, улучшит ли производительность перенос выражения `2 ** X` за пределы циклов? Каким образом вы представили бы это в коде?
- е) Как мы видели в первом упражнении, Python содержит инструмент `map` (функция, список), который также способен генерировать список степеней 2: `map(lambda x: 2 ** x, range(7))`. Попробуйте набрать этот код в интерактивной подсказке; мы опишем `lambda` более формально в следующей части книги, в частности, в главе 19. Поможет ли здесь списковое включение (см. главу 14)?
5. *Сопровождение кода.* Если вы еще таким не занимались, тогда поэкспериментируйте с внесением изменений в код, которые предлагались во врезке “Изменение цветов PyDoc” ранее в главе. Значительный объем работ при создании реального программного обеспечения связан с изменением существующего кода, так что чем раньше вы начнете делать это, тем лучше. Для справки отредактированная мною копия PyDoc находится в пакете примеров для книги по имени `pyurdoc.py`. Чтобы выяснить ее отличия, можете запустить утилиту сравнения файлов (`fc` в Windows) с оригинальным пакетом `pydoc.py` из Python 3.3 (он также включен в состав на случай его радикального изменения в последующих версиях Python). Если PyDoc легче настраивается к моменту чтения этих строк, то настройте цвета согласно текущему соглашению; если процедура предусматривает изменение CSS-файла, то будем надеяться, что она окажется хорошо документированной в руководствах Python.

ЧАСТЬ IV

Функции и генераторы

Основы функций

В части III рассматривались базовые процедурные операторы в Python. Здесь мы перейдем к исследованию набора дополнительных операторов и выражений, которые можно применять для создания собственных функций.

Говоря простыми словами, *функция* является способом группирования набора операторов, позволяющим выполнять их более одного раза в программе — упакованной процедурой, вызываемой по имени. Функции способны вычислять результирующее значение и также дают возможность указывать параметры, которые служат входными данными функции и могут отличаться при каждом выполнении кода. Кодирование операции как функции делает ее в целом полезным инструментом, который можно использовать в разнообразных контекстах.

По существу функции предлагают альтернативу программированию путем *вырезания и вставки* — вместо того, чтобы иметь множество избыточных копий кода операции, мы можем вынести его в единственную функцию. Тем самым мы радикально сокращаем объем будущей работы: если позже операцию потребуется модифицировать, то нам придется вносить изменения только в одиночную копию кода функции, а не в многочисленные копии, разбросанные по всей программе.

Функции также представляют собой наиболее базовую программную структуру Python, которая предназначена для доведения до максимума *многократного использования кода* и подводит нас к более широким понятиям *проектирования* программ. Как будет показано, функции позволяют разбивать сложные системы на поддающиеся управлению части. За счет реализации каждой части в виде функции мы делаем ее многократно применяемой и легкой для кодирования.

В табл. 16.1 приведен обзор основных инструментов, связанных с функциями, которые будут обсуждаться в данной части книги. Сюда входят выражения вызова, два способа создания функций (`def` и `lambda`), два подхода к управлению областью видимости (`global` и `nonlocal`) и два приема передачи результатов обратно в вызывающий код (`return` и `yield`).

Таблица 16.1. Операторы и выражения, связанные с функциями

Оператор или выражение	Примеры
Выражения вызовов	<code>myfunc('spam', 'eggs', meat='ham', *rest)</code>
<code>def</code>	<code>def printer(message):</code> <code> print('Hello ' + message)</code>
<code>return</code>	<code>def adder(a, b=1, *c):</code> <code> return a + b + c[0]</code>

Оператор или выражение	Примеры
global	x = 'old' def changer(): global x; x = 'new'
nonlocal (Python 3.X)	def outer(): x = 'old' def changer(): nonlocal x; x = 'new'
yield	def squares(x): for i in range(x): yield i ** 2
lambda	funcs = [lambda x: x**2, lambda x: x**3]

Для чего используются функции?

Прежде чем углубляться в детали, давайте составим четкое представление о функциях вообще. Функции – это почти универсальная методика структурирования программ. Возможно, вы уже сталкивались с ними в других языках, где они могли называться *подпрограммами* или *процедурами*. В качестве краткого введения функции исполняют две основные роли во время разработки.

Доведение до максимума многократного использования кода и сведение к минимуму избыточности

Как и в большинстве языков программирования, функции Python являются простейшим способом упаковки логики, которую вы, возможно, захотите применять более чем в одном месте и более одного раза. До сих пор весь создаваемый код выполнялся немедленно. Функции позволяют группировать и обобщать код для использования произвольно много раз в более позднее время. Из-за того, что они дают возможность кодировать операцию в одном месте и применять ее во многих местах, функции Python представляют собой самый базовый инструмент *разложения* в языке: они позволяют сократить избыточность кода в программах и посредством этого уменьшить объем работ по сопровождению.

Процедурная декомпозиция

Функции также предлагают инструмент для разбиения систем на части с хорошо определенными ролями. Например, чтобы приготовить пиццу, нужно сначала замесить тесто, раскатать его, добавить начинки, испечь и т.д. Если бы вы программировали робот по приготовлению пиццы, то функции помогли бы разбить общую задачу “приготовления пиццы” на части – по одной функции для каждой подзадачи процесса. Гораздо легче реализовать небольшие подзадачи обособленно, чем сразу полный процесс. В общем случае функции связаны с *процедурой*, которая описывает, как делать что-либо, а не то, в отношении чего делать. Мы увидим, почему такое отличие имеет значение, в части Part VI, когда начнем создавать новые объекты с помощью классов.

В этой части книги мы исследуем инструменты, используемые для написания кода функций в Python: основы функций, правила областей видимости и передачу аргументов вместе с несколькими связанными концепциями, такими как генераторы и инструменты функционального программирования. Мы также возвратимся к введенному ранее в книге понятию полиморфизма, поскольку на данном уровне кодирования его важность становится более очевидной. Вы заметите, что функции не несут в себе особо много нового синтаксиса, но они подводят нас к ряду более развитых идей программирования.

Написание кода функций

Хотя и не очень формально, но в предшествующих главах мы уже применяли несколько функций. Скажем, для создания файлового объекта мы вызывали встроенную функцию `open`; аналогично мы использовали встроенную функцию `len`, чтобы запросить количество элементов в объекте коллекции.

В этой главе мы выясним, как писать *новые* функции в Python. Написанные нами функции ведут себя подобно встроенным функциям, которые мы уже видели: они вызываются в выражениях, принимают значения и возвращают результаты. Но написание новых функций требует приложения ряда дополнительных идей, которые пока еще не были представлены. Кроме того, функции в Python ведут себя совершенно не так, как в компилируемых языках вроде C. Ниже предлагается краткое введение в главные концепции, лежащие в основе функций Python, которые будут исследованы в текущей части книги.

- **`def` является исполняемым кодом.** Функции Python пишутся с помощью нового оператора `def`. В отличие от функций в компилируемых языках, таких как C, `def` представляет собой исполняемый оператор — ваша функция не существует до тех пор, пока Python не встретит и не выполнит `def`. В действительности конечно (и подчас удобно) вкладывать операторы `def` внутрь операторов `if`, циклов `while` и даже других `def`. При типовом применении операторы `def` помещаются в файлы модулей и естественным образом выполняются для генерации функций, когда файл модуля, где они находятся, импортируется в первый раз.
- **`def` создает объект и присваивает его имени.** Когда Python достигает и выполняет оператор `def`, генерируется новый объект функции, который присваивается имени функции. Как и со всеми присваиваниями, имя функции становится ссылкой на объект функции. С именем функции не связано ничего магического — вы увидите, что объект функции можно присваивать другим именам, сохранять в списке и т.п. Объекты функций также могут иметь произвольные определяемые пользователем атрибуты, присоединяемые к ним для регистрации данных.
- **`lambda` создает объект, но возвращает его в качестве результата.** Функции можно также создавать с помощью выражения `lambda` — средства, которое позволяет встраивать определения функций в места, где оператор `def` синтаксически не допускается. Мы отложим рассмотрение этой более сложной концепции до главы 19.
- **`return` отправляет результатирующий объект вызывающему коду.** Когда функция вызывается, вызывающий код приостанавливается до тех пор, пока функция не завершит свою работу и не возвратит управление обратно. Функции, которые вычисляют значение, посылают его вызывающему коду посредством оператора `return`; возвращенное значение становится результатом вызова функции. Оператор `return` без значения просто возвращает управление в вызывающий код (и отправляет стандартный результат `None`).

- **yield** отправляет результатирующий объект вызывающему коду, но запоминает место, где он остановился. В функциях, известных как генераторы, можно также использовать оператор `yield`, чтобы посыпать обратно значение и предохранять их состояние, так что они смогут возобновлять работу позже для производства серии результатов с течением времени. Это еще одна сложная тема, которая раскрывается в последующих главах данной части книги.
- **global** объявляет переменные уровня модуля, предназначенные для присваивания. По умолчанию все имена, присваиваемые в функции, являются локальными для функции и существуют только во время ее выполнения. Чтобы присвоить значение имени из включающего модуля, необходимо указать его в операторе `global` внутри функции. В более общем смысле имена всегда ищутся в областях видимости – местах хранения переменных – и присваивания привязывают имена к областям видимости.
- **nonlocal** объявляет переменные объемлющей функции, предназначенные для присваивания. Подобным образом оператор `nonlocal`, появившийся в Python 3.X, позволяет функции присваивать имя, которое существует в области видимости синтаксически объемлющего оператора `def`. В итоге объемлющие функции могут служить местом сохранения состояния, т.е. информации, запоминаемой между вызовами функции, без потребности в применении разделяемых глобальных имен.
- **Аргументы передаются по присваиванию (по ссылкам на объекты).** В Python аргументы передаются функциям по присваиванию (которое, как вы знаете, означает ссылку на объект). Вы увидите, что в модели Python вызывающий код и функция разделяют объекты по ссылкам, но псевдонимы имен отсутствуют. Изменение имени аргумента внутри функции не приводит к изменению соответствующего имени в вызывающем коде, но модификация на месте переданных изменяемых объектов может изменить объекты, разделяемые с вызывающим кодом, и служит результатом функции.
- **Аргументы передаются по позиции, если только не указано иначе.** Значения, передаваемые вызову функции, по умолчанию сопоставляются с именами аргументов в определении функции слева направо. Ради гибкости вызовам функций можно также передавать аргументы по имени с помощью синтаксиса ключевых слов `имя=значение` и распаковывать произвольно много аргументов для отправки посредством снабжения аргументов звездочками – `*позиционные_аргументы` и `**ключевые_аргументы`. Определения функций используют те же две формы для указания стандартных значений аргументов и сбора произвольно большого количества получаемых аргументов.
- **Аргументы, возвращаемые значения и переменные не объявляются.** Как и абсолютно все в Python, на функции не налагается никаких ограничений по типам. Фактически объявлять заранее о чем-либо, касающемся функции, не нужно: можно передавать аргументы любого типа, возвращать объект любого вида и т.д. Отсюда следует, что единственный вызов функции часто может применяться к разнообразным типам объектов – подойдут любые объекты, которые поддерживают совместимый интерфейс (методы и операции выражений), независимо от их специфических типов.

Если что-то из перечисленного выше оказалось непонятным, то переживать не стоит — в этой части книги мы исследуем все описанные концепции на реальном коде. Давайте начнем с развития ряда упомянутых идей и рассмотрим несколько примеров.

Операторы `def`

Оператор `def` создает объект функции и присваивает его имени. Общий формат `def` выглядит следующим образом:

```
· def имя(аргумент1, аргумент2, ... аргументN):  
    операторы
```

Подобно всем составным операторам Python оператор `def` состоит из строки заголовка, за которой идет блок операторов, обычно с отступом (или одиночный оператор после двоеточия). Блок операторов становится *телом* функции, т.е. кодом, который Python выполняет каждый раз, когда функция позже вызывается.

В строке заголовка `def` указывается *имя* функции, которому присваивается объект функции, а также список из нуля и более *аргументов* (иногда называемых *параметрами*) в круглых скобках. Именам аргументов в заголовке присваиваются объекты, передаваемые в круглых скобках в точке вызова.

Тело функции часто содержит оператор `return`:

```
def имя(аргумент1, аргумент2, ... аргументN):  
    ...  
    return значение
```

Оператор `return` в Python может появляться где угодно в теле функции; по достижении он заканчивает вызов функции и посыпает результат обратно вызывающему коду. Оператор `return` состоит из необязательного выражения с объектным значением, которое дает результат функции. Если значение опущено, тогда `return` отправляет обратно `None`.

Оператор `return` сам по себе также необязателен; если он отсутствует, то выход из функции происходит, когда поток управления достигает конца тела функции. Формально функция без оператора `return` автоматически возвращает объект `None`, но такое возвращаемое значение обычно при вызове игнорируется.

Функции могут также содержать операторы `yield`, которые предназначены для производства серии значений с течением времени, но мы отложим их обсуждение до исследования темы генераторов в главе 20.

Оператор `def` исполняется во время выполнения

Оператор `def` в Python является подлинным исполняемым оператором: при выполнении он создает новый объект функции и присваивает его имени. (Помните, что в Python мы имеем лишь *исполняющую среду*; понятие отдельного времени компиляции не существует.) Поскольку `def` — оператор, он может появляться везде, где допускается оператор, даже внутри других операторов. Например, хотя операторы `def` обычно выполняются, когда включающий их модуль импортируется, также совершенно законно вкладывать `def` внутрь оператора `if` для выбора между альтернативными определениями функций:

```
if test:  
    def func():    # Одно определение func  
    ...
```

```
else:  
    def func():           # Иначе другое определение func  
    ...  
    ...  
func()                      # Вызов выбранной и созданной версии
```

Один из способов понять этот код — осознать, что оператор `def` во многом похож на оператор `=`: он просто присваивает имя во время выполнения. В отличие от компилируемых языков вроде С функции Python не нуждаются в полном определении перед запуском программы. В более общем плане операторы `def` не оцениваются до тех пор, пока не будут достигнуты и выполнены, и код *внутри* операторов `def` не выполняется до вызова функций.

Из-за того, что определение функций происходит во время выполнения, в имени функции нет ничего особенного. Важен объект, на который оно ссылается:

```
othername = func          # Присваивание объекта функции  
othername()                # Снова вызывает func
```

Здесь функция была присвоена другому имени и вызвана через него. Как и все остальное в Python, функции — это просто *объекты*; они явно записываются в память во время выполнения программы. На самом деле кроме вызовов функции позволяют присоединять произвольные атрибуты для регистрации информации с целью более позднего использования:

```
def func(): ...             # Создание объекта функции  
func()                      # Вызов объекта функции  
func.attr = value           # Присоединение атрибутов
```

Первый пример: определения и вызовы

Не считая таких концепций времени выполнения (которые кажутся наиболее уникальными в основном программистам с опытом работы на традиционных компилируемых языках), функции Python прямолинейны в применении. Давайте напишем код первого реального примера, чтобы продемонстрировать основы. Вы увидите, что у функции есть две стороны: *определение* (оператор `def`, создающий функцию) и *вызов* (выражение, которое сообщает Python о необходимости выполнения тела функции).

Определение

Ниже показано интерактивно набранное определение функции по имени `times`, которое возвращает произведение двух аргументов:

```
>>> def times(x, y):      # Создание и присваивание функции  
...     return x * y        # Тело выполняется при вызове  
...
```

Когда Python достигает и выполняет этот оператор `def`, он создает новый объект функции, умещающий в себе код функции, и присваивает его имени `times`. Обычно такой оператор находится в файле модуля и выполняется при его импортировании; однако для чего-нибудь небольшого вполне достаточно интерактивной подсказки.

Вызов

Оператор `def` создает функцию, но не вызывает ее. После выполнения `def` функцию можно вызывать (выполнять) в своей программе, добавляя к имени функции

круглые скобки. Круглые скобки могут дополнительно содержать один и более объектов-аргументов, подлежащих передаче (присваиванию) именам в заголовке функции:

```
>>> times(2, 4)          # Аргументы в круглых скобках
8
```

Выражение вызова передает в `times` два аргумента. Как упоминалось ранее, аргументы передаются по присваиванию, так что в рассматриваемом случае имени `x` в заголовке функции присваивается значение `2`, у присваивается значение `4` и тело функции выполняется. Телом функции `times` является всего лишь оператор `return`, который отправляет обратно результат в качестве значения выражения вызова. Возвращаемый объект был выведен интерактивно (как и в большинстве языков, в Python значением `2 * 4` будет `8`), но если объект необходимо использовать позже, тогда его можно присвоить переменной. Вот пример:

```
>>> x = times(3.14, 4)    # Сохранение результирующего объекта
>>> x
12.56
```

Давайте посмотрим, что происходит, когда функция вызывается с передачей объектов очень разных видов:

```
>>> times('Ni', 4)        # Функции "лишены типов"
'NiNiNiNi'
```

Теперь наша функция означает нечто совсем другое (здесь на ум приходит кинофильм “Монти Пайтон: а теперь нечто совсем другое”). В последнем вызове `x` и `y` вместо двух чисел передаются строка и целое число. Вспомните, что операция `*` работает с числами и последовательностями; поскольку в Python мы никогда не объявляем типы переменных, аргументов или возвращаемых значений, то в состоянии применять функцию `times` либо для умножения чисел, либо для повторения последовательностей.

Другими словами, смысл и работа функции `times` зависят от того, что мы ей передаем. В этом и заключается основная идея (и вероятно побудительная причина использовать язык), которая заслуживает некоторой детализации.

Полиморфизм в Python

Как только что выяснилось, сам смысл выражения `x * y` в нашей простой функции `times` зависит целиком от видов объектов, указываемых для `x` и `y`; таким образом, также самая функция способна выполнять умножение в одной ситуации и повторение в другой. Python оставляет на усмотрение *объектов* делать что-то разумное с точки зрения синтаксиса. Вообще говоря, операция `*` представляет собой всего лишь координирующий механизм, который передает контроль обрабатываемым объектам.

Поведение с зависимостью от типов подобного рода известно как *полиморфизм* – термин, впервые упомянутый в главе 4, который по существу означает, что смысл операции зависит от обрабатываемых ею объектов. Из-за того, что Python является динамически типизируемым языком, полиморфизм в нем буквально процветает. Фактически *каждая* операция в Python обладает полиморфизмом: вывод, индексация, операция `*` и многие другие.

Такое поведение неслучайно и в большой степени объясняет лаконичность и гибкость языка. Например, единственная функция обычно может применяться к целой категории типов объектов автоматически. До тех пор, пока объекты поддерживают ожидаемый *интерфейс* (он же протокол), функция в состоянии обрабатывать их.

То есть когда передаваемые в функцию объекты имеют ожидаемые методы и операции выражений, они считаются автоматически совместимыми с логикой функции.

Даже в нашей простой функции `times` это означает, что *любые* два объекта, которые поддерживают операцию `*`, будут работать независимо от того, что они собой представляют и когда закодированы. Функция `times` будет работать с двумя числами (выполняя умножение), со строкой и числом (выполняя повторение) или с любой другой комбинацией объектов, поддерживающих ожидаемый интерфейс — даже с объектами, основанными на классах, которые мы даже еще не придумали.

Более того, если переданные объекты *не* поддерживают ожидаемый интерфейс, то Python обнаружит ошибку при попытке выполнить выражение `*` и автоматически сгенерирует исключение. Следовательно, обычно не имеет смысла писать код проверки на предмет ошибок. На самом деле добавление такого кода проверки ограничило бы полезность нашей функции, поскольку она смогла бы работать только с объектами, чьи типы проверяются.

Описанное поведение оказывается ключевым философским отличием между Python и статически типизированными языками вроде C++ и Java: в Python ваш код *не обязан заботиться о специфических типах данных*. Если это произойдет, тогда код ограничится работой только с теми типами, которые вы предвидели при его написании, и не будет поддерживать другие совместимые типы объектов, которые возможно появятся в будущем. Хотя можно проверять на предмет принадлежности к типам с помощью инструментов, подобных встроенной функции `type`, наличие таких проверок сведет на нет гибкость кода. Вообще говоря, в Python мы пишем код для *интерфейсов* объектов, а не для типов данных¹.

Разумеется, определенные программы имеют уникальные требования, и суть такой полиморфной модели программирования в том, что мы обязаны тестируировать свой код с целью выявления ошибок, а не предоставлять компилятору объявления типов, которые он сможет использовать для обнаружения специфических разновидностей ошибок заблаговременно. Тем не менее, в обмен на небольшое первоначальное тестирование мы получаем радикальное сокращение объема кода, который приходится писать, и весомое увеличение гибкости результирующего кода. Как вы увидите, на практике это означает чистый выигрыш.

Второй пример: пересечение последовательностей

Давайте рассмотрим второй пример функции, которая делает что-то более полезное, чем перемножение аргументов, и дополнительно иллюстрирует основы функций.

В главе 13 мы писали код цикла `for`, который накапливал элементы, общие для двух строк. Там мы отметили, что код получился не настолько полезным, каким мог бы быть, поскольку он настроен на работу только со специфическими переменными

¹ В последние годы такое полиморфное поведение также стало известным под названием *универсальная типизация*, которая гласит, что ваш код не обязан заботиться о том, является ли объект *уткой*, а только о том, что он *крякает*. Подойдет все, что крякает, утка или нет, и реализация кряканья возлагается на объект — принцип станет гораздо более ясным, когда мы приступим к изучению классов в части VI. Конечно, это является лишь наглядной метафорой, хотя на самом деле представляет собой просто новое обозначение старой идеи, и сценарии использования для крякающего программного обеспечения могут показаться ограниченными в материальном мире (говорят он, разгребая почтовые завалы от активистов-орнитологов...).

и не может быть повторно выполнен позже. Конечно, мы могли бы скопировать код и вставить во все места, где его необходимо выполнять, но такое решение нельзя оценить как хорошее или универсальное. Нам по-прежнему пришлось бы редактировать каждую копию для поддержки разных имен последовательностей, а изменение алгоритма потребовало бы тогда внесения изменений в многочисленные копии.

Определение

К этому времени, вероятно, вы уже догадались, что выход из затруднительного положения предусматривает упаковку цикла `for` внутрь функции. Такое решение обладает некоторыми преимуществами.

- Помещение кода в функцию делает его инструментом, который можно запускать столько раз, сколько нужно.
- Поскольку вызывающий код может передавать произвольные аргументы, функции достаточно универсальны для того, чтобы работать с любыми двумя последовательностями (или другими итерируемыми объектами), пересечение которых необходимо получить.
- Когда логика упакована внутрь функции, в случае изменения реализации пересечения код придется модифицировать только в одном месте.
- Помещение кода функции в файл модуля означает, что ее можно импортировать и многократно применять в любой программе, выполняемой на компьютере.

В действительности помещение кода в функцию превращает ее в универсальную утилиту пересечения:

```
def intersect(seq1, seq2):  
    res = []                      # Начать с пустого результата  
    for x in seq1:                # Просмотр seq1  
        if x in seq2:            # Общий элемент?  
            res.append(x)        # Добавить в конец результата  
    return res
```

Трансформация простого кода из главы 13 в приведенную выше функцию прямолинейна; мы просто помещаем первоначальную логику ниже заголовка `def` и делаем объекты, на которых она оперирует, передаваемыми именами параметров. Из-за того, что функция вычисляет результат, мы также добавляем оператор `return` для отправки результирующего объекта обратно вызывающему коду.

Вызов

Прежде чем можно будет вызывать функцию, ее потребуется создать. Необходимо выполнить оператор `def` функции, для чего либо набрать его в интерактивной подсказке, либо поместить в файл модуля и импортировать модуль. После выполнения `def` функцию можно вызывать, передавая ей любые два объекта последовательностей в круглых скобках:

```
>>> s1 = "SPAM"  
>>> s2 = "SCAM"  
>>> intersect(s1, s2)          # Строки  
['S', 'A', 'M']
```

Здесь мы передаем две строки и получаем обратно список, содержащий общие символы. Алгоритм, используемый функцией, прост: “для каждого элемента в первом

аргументе выполнить проверку, содержится ли он также во втором аргументе, и если да, то добавить его в результат". На языке Python алгоритм выглядит короче, чем на естественном языке, но смысл остается тем же.

Ради справедливости следует отметить, что наша функция пересечения работает довольно медленно (она выполняет вложенные циклы), не является настоящим математическим пересечением (в результате могут присутствовать дубликаты) и не требуется вообще (тип множества Python предоставляет встроенную операцию пересечения). На самом деле функцию `intersect` можно было бы заменить выражением спискового включения, т.к. она демонстрирует классический кодовый шаблон цикла с накоплением данных:

```
>>> [x for x in s1 if x in s2]
['S', 'A', 'M']
```

Однако в качестве примера, иллюстрирующего основы функций, он вполне подходит – как будет объясняться в следующем разделе, этот одиночный фрагмент можно применять к целому диапазону типов объектов. После дополнительного исследования режимов передачи аргументов в главе 18 мы улучшим и расширим его для поддержки произвольно большого количества операндов.

Еще раз о полиморфизме

Как и все добродорядочные функции в Python, функция `intersect` полиморфна, т.е. работает на произвольных типах при условии, что они поддерживают ожидаемый интерфейс объекта:

```
>>> x = intersect([1, 2, 3], (1, 4))      # Разнородные типы
>>> x                                     # Сохраненный результирующий объект
[1]
```

В данном примере мы передали нашей функции объекты разных типов – список и кортеж (разнородные типы) – и она по-прежнему выбрала общие элементы. Поскольку нет необходимости указывать типы аргументов заранее, функция `intersect` благополучно проходит по передаваемым объектам последовательностей любого вида, пока они поддерживают ожидаемые интерфейсы.

Для функции `intersect` это означает, что первый аргумент должен поддерживать цикл `for`, а второй – проверку членства `in`. Любые два таких объекта будут работать независимо от их специфических типов: физически хранимые последовательности наподобие строк и списков; все итерируемые объекты из главы 14, включая файлы и словари; и даже любые объекты, основанные на написанных нами классах, которые применяют методики перегрузки операций, обсуждаемые позже в книге².

² Этот код всегда будет работать, если мы выполняем пересечение содержимого файлов, которое получается посредством `file.readlines()`. Тем не менее, в зависимости от реализации операции `in` или универсальной итерации файловым объектом код может не работать для пересечения строк в открытых входных файлах напрямую. В общем случае после того, как однажды был достигнут конец файла, файлы должны быть позиционированы в начало (например, с помощью вызова `file.seek(0)` или еще одного `open`), и потому они являются однопроходными итераторами. Как будет показано в главе 30, где исследуется перегрузка операций, объекты реализуют операцию `in` либо путем предоставления специфического метода `__contains__`, либо за счет поддержки универсального протокола итерации посредством метода `__iter__` или более старого `__getitem__`; классы могут кодировать упомянутые методы произвольным образом, определяя в итоге смысл итерации для своих данных.

И снова, когда мы передаем объекты, которые не поддерживают такие интерфейсы (скажем, числа), Python автоматически обнаруживает несоответствие и генерирует исключение — именно то, что нам нужно, и лучшее, что мы могли бы сделать самостоятельно, если бы явно записывали проверки на предмет типов. Не записывая проверки на предмет типов и позволяя Python обнаруживать несоответствия, мы сокращаем объем кода, подлежащего написанию, и увеличиваем гибкость итогового кода.

Локальные переменные

Пожалуй, самой интересной частью рассмотренного примера являются его имена. Дело в том, что переменная `res` внутри `intersect` представляет собой так называемую *локальную переменную* — имя, которое видно только в коде внутри оператора `def` функции и существует лишь на период выполнения функции. Фактически из-за того, что все имена, тем или иным образом *присваиваемые* внутри функции, по умолчанию классифицируются как локальные переменные, почти все имена в функции `intersect` оказываются локальными переменными:

- переменной `res` производится очевидное присваивание, так что она локальная;
- аргументы передаются по присваиванию, поэтому `seq1` и `seq2` тоже локальные;
- цикл `for` присваивает элементы переменной, а потому имя `x` — также локальная переменная.

Все перечисленные локальные переменные начинают свое существование, когда функция вызывается, и исчезают при выходе из функции — оператор `return` в конце `intersect` посылает обратно результирующий *объект*, но имя `res` прекращает существовать. По этой причине переменные функции не запоминают свои значения между вызовами; хотя возвращаемый функцией объект остается, сохранение прочих видов информации о состоянии требует других методик. Однако для полного понимания локальных переменных и состояния нам необходимо перейти к исследованию областей видимости в главе 17.

Резюме

В главе были представлены ключевые идеи, лежащие в основе определения функций: синтаксис и работа операторов `def` и `return`, поведение выражений вызова функций, а также понятие и преимущества полиморфизма в функциях Python. Вы узнали, что оператор `def` является исполняемым кодом, который создает объект функции во время выполнения. Когда функция позже вызывается, объекты передаются ей по присваиванию (вспомните, что присваивание в Python подразумевает ссылку на объект и внутренне означает указатель, как объяснялось в главе 6), а вычисленные значения отправляются обратно оператором `return`. Мы также начали исследование концепций локальных переменных и областей видимости, но отложили более подробное рассмотрение данных тем до главы 17. А теперь закрепите полученные знания, ответив на контрольные вопросы главы.

Проверьте свои знания: контрольные вопросы

1. В чем смысл написания функций?
2. В какой момент Python создает функцию?
3. Что функция возвращает, если в ней нет ни одного оператора `return`?
4. Когда выполняется код, вложенный внутрь оператора определения функции?
5. Что не так с проверкой типов объектов, передаваемых в функцию?

Проверьте свои знания: ответы

1. Функции – это наиболее базовый способ избегания избыточности кода в Python – вынесение кода в функции означает наличие только одной копии кода операции, который возможно придется обновлять в будущем. Функции также являются базовой единицей многократного использования кода в Python – помещение кода в функции делает его многократно применяемым инструментом, допускающим вызов в разнообразных программах. Наконец, функции позволяют разбивать сложную систему на поддающиеся управлению части, каждая из которых может разрабатываться по отдельности.
2. Функция создается, когда Python достигает оператора `def`; данный оператор создает объект функции и присваивает его имени функции. Обычно это происходит при импортировании файла модуля, содержащего `def`, другим модулем (вспомните, что импорт модуля приводит к выполнению кода в файле от начала до конца, включая любые операторы `def`), но может также случаться, когда `def` набирается в интерактивной подсказке или вкладывается в другие операторы, такие как `if`.
3. По умолчанию функция возвращает объект `None`, если поток управления добрался до конца тела функции, не встретив ни одного оператора `return`. Такие функции обычно вызываются посредством операторов выражений, поскольку присваивание переменным их результатов `None` в целом бессмысленно. Оператор `return` без выражения в нем также возвращает `None`.
4. Тело функции (код, вложенный внутрь оператора определения функции) выполняется, когда функция впоследствии вызывается с помощью выражения вызова. При каждом вызове функции ее тело выполняется заново.
5. Проверка типов объектов, передаваемых функции, фактически уничтожает гибкость кода, ограничивая функцию работой только со специфическими типами. Без таких проверок функция вполне вероятно была бы способной обрабатывать целый диапазон типов объектов – любых объектов, которые поддерживают интерфейс, ожидаемый функцией. (Термин интерфейс означает набор методов и операций выражений, выполняемых кодом функции.)

Области видимости

В главе 16 были представлены основы определения и вызова функций. Вы видели, что центральная модель функций Python проста в применении, но даже элементарные примеры функций быстро приводят нас к вопросам о назначении переменных в коде. В этой главе мы займемся деталями *областей видимости* Python — мест, где переменные определяются и ищутся. Подобно файлам модулей области видимости помогают предотвратить конфликты имен в коде программы: имена, определяемые в одной программной единице, не пересекаются с именами в другой.

Как будет показано, место в коде, где происходит присваивание имени, критически важно при определении назначения имени. Также обнаружится, что использование областей видимости может оказывать большое воздействие на работу по сопровождению программы; например, чрезмерное употребление *глобальных переменных* в целом является плохой практикой. С другой стороны, области видимости способны предоставить способ сохранения информации о состоянии между вызовами функции, а в определенных ролях выступают в качестве альтернативы классам.

Основы областей видимости в Python

Вы уже готовы приступить к написанию собственных функций, но вам необходимо получить более формальное представление о назначении имен в Python. В случае использования имени в программе Python создает, изменяет или ищет имя внутри того, что называется *пространством имен* — месте, где имена существуют. Когда речь идет о поиске значения имени применительно к коду, термин *область видимости* относится к пространству имен: т.е. местоположение присваивания имени в исходном коде определяет область, где имя является видимым для кода.

Почти все, что связано с именами, включая классификацию областей видимости, в Python происходит во время присваивания. Как мы уже видели, имена в Python начинают свое существование, когда им впервые присваиваются значения, и чтобы имена можно было использовать, им должны быть присвоены значения. Поскольку имена заранее не объявляются, Python применяет местоположение присваивания имени для ассоциирования (т.е. *связывания*) со специфическим пространством имен. Другими словами, место присваивания имени в исходном коде определяет пространство имен, в котором оно будет существовать, и отсюда его область видимости.

Помимо упаковки кода для многократного использования функции добавляют к программам дополнительный уровень, чтобы свести к минимуму возможность возникновения конфликтов между переменными с одним и тем же именем — *по умолчанию все*

имена, которым выполнено присваивание внутри функции, ассоциируются с пространством имен данной функции и никаким другим. Вот что указанное правило означает.

- Имена, присвоенные внутри `def`, могут быть видны только в коде внутри этого оператора `def`. Ссыльаться на такие имена извне функции нельзя.
- Имена, присвоенные внутри `def`, не конфликтуют с переменными за пределами `def`, даже если те же самые имена применяются где-то в другом месте. Имя `X`, присвоенное вне заданного оператора `def` (т.е. в другом `def` или на верхнем уровне файла модуля), представляет собой переменную, совершенно отличающуюся от имени `X`, присвоенную внутри этого `def`.

Во всех случаях область видимости переменной (где она может использоваться) всегда определяется местом ее присваивания в исходном коде и не имеет никакого отношения к тому, какая функция какую вызывает. На самом деле, как будет объясняться позже в главе, переменные могут присваиваться в трех разных местах, соответствующих трем разным областям видимости:

- если переменная присваивается внутри `def`, то она будет локальной в этой функции;
- если переменная присваивается в объемлющем `def`, тогда она будет нелокальной в отношении вложенных функций;
- если переменная присваивается за пределами всех `def`, то она будет глобальной в целом файле.

Мы называем это *лексической областью видимости*, потому что области видимости переменных определяются полностью местоположениями переменных в файлах исходного кода программы, а не вызовами функций.

Например, в следующем файле модуля присваивание `X = 99` создает *глобальную* переменную по имени `X` (видимую в данном файле), но присваивание `X = 88` создает *локальную* переменную `X` (видимую только внутри оператора `def`):

```
X = 99      # X с глобальной областью видимости (модуль)
def func():
    X = 88  # X с локальной областью видимости (функция): другая переменная
```

Несмотря на то что обе переменные имеют имя `X`, связанные с ними области видимости делают их разными. Совокупный эффект заключается в том, что области действия функций помогают избежать конфликтов имен в программах и сделать функции более самодостаточными программными единицами — в их коде не придется беспокоиться об именах, применяемых в других местах программы.

Детали, касающиеся областей видимости

До того, как мы начали писать функции, весь создаваемый код располагался на верхнем уровне модуля (т.е. не был вложенным в `def`), в результате чего используемые имена существовали либо в самом модуле, либо во встроенных компонентах, предварительно определенных Python (например, `open`). Формально интерактивная подсказка — это модуль по имени `__main__`, который выводит результаты и не сохраняет свой код; во всех остальных отношениях он похож на верхний уровень файла модуля.

Однако функции предоставляют вложенные пространства имен (области видимости), которые локализуют применяемые в них имена, так что имена внутри функции не конфликтуют с именами за ее пределами (в модуле или в другой функции). Функции

определяют локальную область видимости, а модули — глобальную область видимости со следующими свойствами.

- Включающий модуль является глобальной областью видимости. Каждый модуль представляет собой глобальную область видимости, т.е. пространство имен, в котором существуют переменные, создаваемые (присваиваемые) на верхнем уровне файла модуля. Для внешнего мира глобальные переменные становятся атрибутами объекта модуля после его импортирования, но могут также использоваться в качестве простых переменных внутри самого файла модуля.
- Глобальная область видимости охватывает только одиночный файл. Не обманывайтесь насчет слова “глобальная” — имена на верхнем уровне файла глобальны только в коде внутри этого одного файла. Вообще говоря, в Python нет понятия единственной всеобъемлющей глобальной области видимости, основанной на файлах. Взамен имена распределяются по модулям, и модуль должен всегда явным образом импортироваться, чтобы появилась возможность работы с именами, определенными в его файле. Когда вы слышите “глобальный” в Python, подразумевайте “модуль”.
- Присвоенные имена являются локальными, если только не объявлены глобальными или нелокальными. По умолчанию все имена, присвоенные внутри определения функции, помещаются в локальную область видимости (пространство имен, ассоциированное с вызовом функции). Если необходимо присвоить имя, которое существует на верхнем уровне модуля, включающего в себя функцию, тогда такое имя можно объявить как глобальное в операторе `global` внутри функции. Если нужно присвоить имя, находящееся в объемлющем операторе `def`, то начиная с Python 3.X, имя можно объявить как нелокальное в операторе `nonlocal`.
- Все остальные имена представляют собой локальные, глобальные или встроенные в объемлющей функции. Предполагается, что имена, которым не присваиваются значения в определении функции, являются локальными именами в объемлющей области видимости, которые определены в окружающем операторе; глобальными именами, существующими в пространстве имен включающего модуля; или встроенными именами в предварительно определенном модуле `builtins`, предоставляемом Python.
- Каждый вызов функции создает новую локальную область видимости. Всякий раз, когда вы вызываете функцию, создается новая локальная область видимости, т.е. пространство имен, в котором обычно будут существовать имена, созданные внутри функции. Вы можете думать о каждом операторе `def` (и выражении `lambda`) как об определении новой локальной области видимости, но локальная область видимости в действительности соответствует вызову функции. Поскольку Python разрешает функциям вызывать самих себя в цикле (расширенный прием, известный как рекурсия и кратко упомянутый в главе 9 при исследовании сравнений), каждый активный вызов получает собственную копию локальных переменных функции. Рекурсия также полезна в функциях, написанных для обработки структур, форму которых нельзя предугадать заранее; мы исследуем ее более полно в главе 19.

Здесь уместно отметить несколько тонкостей. Прежде всего, имейте в виду, что имена в коде, набранном в интерактивной подсказке, тоже существуют в модуле и подчиняются обычным правилам областей видимости: они являются глобальными пере-

менными, которые доступны целому интерактивному сеансу. Модули будут подробно рассматриваться в следующей части книги.

Также обратите внимание, что *присваивание любого вида* внутри функции классифицирует имя как локальное, включая операторы `=`, имена модулей в операторе `import`, имена функций в операторе `def`, имена аргументов функций и т.д. Присваивание имени любым способом внутри `def` по умолчанию делает его локальным в данной функции.

Наоборот, *изменения на месте* объектов не классифицируют имена как локальные; так поступают только присваивания действительных имен. Скажем, если имени `L` присваивается список на верхнем уровне модуля, то оператор `L = X` внутри функции будет классифицировать `L` как локальное имя, но `L.append(X)` – нет. В последнем случае мы изменяем списоковый объект, на который ссылается `L`, а не само имя `L` – переменная `L` находится в глобальной области видимости, как обычно, и Python успешно модифицирует ее, не требуя объявления `global` (или `nonlocal`). Как всегда, это помогает сохранять четкое различие между именами и объектами: изменение объекта не является присваиванием имени.

Распознавание имен: правило LEGB

Несмотря на то, что объяснения в предыдущем разделе могут сбивать с толку, на самом деле все сводится к трем простым правилам. Внутри оператора `def`:

- присваивания имен по умолчанию создают либо изменяют локальные имена;
- ссылки на имена производят поиск самое большое в четырех областях видимости – в локальной области видимости, затем в областях видимости объемлющих функций (если есть), далее в глобальной области видимости и, наконец, во встроенной области видимости;
- имена, объявленные в операторах `global` и `nonlocal`, отображают присвоенные имена на области видимости включающего модуля и функции соответственно.

Другими словами, все имена, присвоенные внутри оператора `def` функции (или внутри `lambda` – выражение, которое мы исследуем позже), по умолчанию будут локальными. В функциях можно свободно использовать имена, присвоенные в синтаксически объемлющих функциях и глобальной области видимости, но для изменения таких имен они должны быть объявлены как нелокальные и глобальные.

Ниже описана схема распознавания имен Python, которая иногда называется *правилом LEGB* согласно названиям областей видимости.

- Когда внутри функции указывается неуточненное имя, Python ищет его максимум в четырех местах – в локальной (`L` (`local`)) области видимости, затем в локальных областях видимости любых объемлющих (`E` (`enclosing`)) операторов `def` и `lambda`, далее в глобальной (`G` (`global`)) области видимости и, наконец, во встроенной (`B` (`built-in`)) области видимости – и останавливает поиск на первом же месте, где обнаруживается имя. Если в результате такого поиска имя найти не удалось, тогда Python сообщит об ошибке
- Когда внутри функции выполняется присваивание имени (вместо просто ссылки на него в выражении), Python всегда создает либо изменяет имя в локальной области видимости, если только оно не было объявлено в этой функции как глобальное или нелокальное.
- Когда производится присваивание имени за пределами всех функций (т.е. на верхнем уровне файла модуля или в интерактивной подсказке), локальная область видимости совпадает с глобальной – пространством имен модуля.

Поскольку имена должны быть присвоены, прежде чем их можно будет использовать (как объяснялось в главе 6), в данной модели ничего не происходит автоматически: присваивания всегда однозначно определяют области видимости.

На рис. 17.1 проиллюстрированы четыре области видимости Python. Обратите внимание, что уровню поиска второй области видимости (*E* – области видимости объемлющих `def` или `lambda`) формально может соответствовать сразу несколько уровней поиска. Такая ситуация возникает, когда одни функции вкладываются внутрь других функций, и расширяется оператором `nonlocal` в Python 3.X¹.

Встроенная область видимости (Python)

Имена, предварительно присвоенные
в модуле `builtins`: `open`, `range`, `SyntaxError`...

Глобальная область видимости (модуль)

Имена, присвоенные на верхнем уровне файла модуля
или объявленные глобальными в операторе `def` внутри файла.

Локальные области видимости объемлющих функций

Имена в локальной области видимости любых
объемлющих функций (`def` или `lambda`),
от самой внутренней до наружной.

Локальная область видимости (функция)

Имена, так или иначе присвоенные внутри функции (`def` или `lambda`)
и не объявленные глобальными в этой функции.

Рис. 17.1. Правило поиска в областях видимости LGB. Когда производится ссылка на переменную, Python ищет ее в следующем порядке: в локальной области видимости, в локальных областях видимости объемлющих функций, в глобальной области видимости и во встроенной области видимости. Поиск прекращается при первом нахождении. Обычно область видимости переменной определяется местом в коде, где она присваивается. Объявления `nonlocal` в Python 3.X также способны принудительно отображать имена на области видимости объемлющих функций независимо от того, были имена присвоены или нет

Также имейте в виду, что приведенные правила применяются только к простым именам *переменных* (например, `spam`). В частях V и VI вы увидите, что уточненные имена *атрибутов* (скажем, `объект.spam`) существуют в индивидуальных объектах и следуют совершенно другому набору правил поиска, нежели тот, который был раскрыт здесь. Ссылки на имена атрибутов, указываемые после точек (`.`), приводят к поиску в одном или большем количестве *объектов*, а не в областях видимости, и фактически

¹ В первом издании книги правило поиска в областях видимости называлось “правилом LGB”. Уровень *E* объемлющего оператора `def` появился в Python позже, чтобы избавиться от необходимости в явной передаче имен из объемлющей области видимости с помощью стандартных аргументов — тема, вызывающая обычно незначительный интерес у новичков в Python, которую мы рассмотрим позже в книге. Так как теперь для этой области видимости предназначен оператор `nonlocal` в Python 3.X, правило поиска в наши дни вероятно лучше было бы называть “LNGB”, но обратная совместимость важна и в книгах. Нынешняя форма данной аббревиатуры также не учитывает более новые и малоизвестные области видимости ряда включений и обработчиков исключений, но аббревиатуры длиннее четырех букв сводят на нет их предназначение!

могут задействовать то, что в модели объектно-ориентированного программирования Python называется *наследованием*; о нем пойдет речь в части VI книги.

Другие области видимости Python: обзор

Хотя в текущем месте книги это не особенно ясно, формально в Python есть еще три области видимости – временные переменные циклов в ряде включений, переменные ссылок на исключения в некоторых обработчиках `try` и локальные области видимости в операторах `class`. Первые две являются особыми случаями, редко влияющими на реальный код, а третья подпадает под действие правила LEGB.

Большинство блоков операторов и прочих конструкций не локализуют имена, используемые внутри них, кроме перечисленных ниже ситуаций, специфичных к версиям (их переменные не доступны в окружающем коде, но не конфликтуют с ним, и затрагивают темы, которые раскрываются позже в книге).

- Переменные включений – переменная `X`, применяемая для ссылки на элемент текущей итерации в выражении включения вроде `[X for X in I]`. Поскольку они могут конфликтовать с другими именами и отражать внутреннее состояние в генераторах, в Python 3.X такие переменные являются локальными для самого выражения во всех формах включений: генератора, списка, множества и словаря. В Python 2.X они локальны в генераторных выражениях, а также во включениях множеств и словарей, но не в списковых включениях, которые отображают свои имена на область видимости за пределами выражения. Напротив, операторы цикла `for` никогда не локализуют свои переменные внутри блока операторов в любой версии Python. За дополнительными деталями и примерами обращайтесь в главу 20.
- Переменные исключений – переменная `X`, используемая для ссылки на сгенерированное исключение в конструкции обработчика оператора `try` наподобие `except E as X`. Из-за того, что они могут задерживать освобождение памяти при сборке мусора, в Python 3.X такие переменные являются локальными в этом блоке `except` и на самом деле уничтожаются при выходе из блока (даже если они применялись ранее в коде!). В Python 2.X указанные переменные продолжают существовать после оператора `try`. Дополнительные сведения приведены в главе 34.

Описанные контексты дополняют правило LEGB, а не модифицируют его. Например, присвоенные во включении, просто связываются с добавочной особой областью видимости; другие имена, на которые производятся ссылки внутри таких выражений, следуют обычному правилу поиска LEGB.

Полезно отметить, что рассматриваемый в части VI оператор `class` тоже создает новую *локальную* область видимости для имен, присваиваемых внутри верхнего уровня своего блока. Что касается `def`, то присваиваемые внутри `class` имена не конфликтуют с именами в других местах и подчиняются правилу поиска LEGB, где блок `class` относится к уровню L. Подобно модулям и импортированию после окончания операторов `class` такие имена также превращаются в атрибуты объектов класса.

Тем не менее, в отличие от функций имена `class` не создаются посредством вызова: обращения к объектам класса генерируют *экземпляры*, которые наследуют имена, присвоенные внутри `class`, и записывают объектное состояние в виде атрибутов. В главе 29 будет показано, что хотя правило LEGB используется для распознавания имен на верхнем уровне класса и на верхнем уровне вложенных в него функций методов, при поиске областей видимости сами классы *пропускаются* – их имена должны

извлекаться как атрибуты объектов. Поскольку Python ищет указанные имена в объемлющих функциях, но не в объемлющих классах, правило LEGB по-прежнему применимо к объектно-ориентированному коду.

Пример области видимости

Давайте разберем более крупный пример, который демонстрирует идеи, лежащие в основе областей видимости. Предположим, что в файле модуля находится следующий код:

```
# Глобальная область видимости
X = 99                      # Имена X и func присваиваются в модуле: глобальные

def func(Y):                  # Имена Y и Z присваиваются в функции: локальные
    # Локальная область видимости
    Z = X + Y                # Имя X является глобальным
    return Z

func(1)                       # Имя func в модуле: result=100
```

Для выполнения своей работы модуль и содержащаяся в нем функция задействуют несколько имен. Используя правила областей видимости Python, мы можем классифицировать имена, как описано далее.

Глобальные имена: X, func

Имя X является глобальным, потому что оно присваивается на верхнем уровне файла модуля; на него можно ссылаться внутри функции как на простую неуточненную переменную, не объявляя глобальным. Имя func глобально по той же причине; оператор def присваивает объект функции имени func на верхнем уровне модуля.

Локальные имена: Y, Z

Имена Y и Z являются локальными в функции (и существуют только в период ее выполнения), т.к. им обоим присваиваются значения в определении функции: Z посредством оператора =, а Y из-за того, что аргументы всегда передаются по присваиванию.

Логическое обоснование такой схемы отделения имен состоит в том, что локальные переменные служат *временными* именами, которые необходимы лишь в течение выполнения функции. Скажем, в предыдущем примере аргумент Y и результат сложения Z существуют только внутри функции; упомянутые имена не пересекаются с пространством имен вмещающего модуля (или любой другой функции, если уж на то пошло). В действительности по завершении вызова функции локальные переменные удаляются из памяти, а объекты, на которые они ссылаются, могут быть подвергнуты *сборке мусора*, если на них нет ссылок где-то в другом месте. Это автоматический внутренний шаг, но он помогает минимизировать расход памяти.

Разграничение локальные/глобальные также облегчает понимание функций, т.к. большинство имен, используемых функцией, возникает в самой функции, а не каком-то произвольном месте модуля. Кроме того, поскольку вы можете быть уверены в том, что локальные имена не будут изменяться другой функцией в программе, они упрощают отладку и модификацию программ. Функции представляют собой самодостаточные программные единицы.

Встроенная область видимости

Мы уже абстрактно обсуждали встроенную область видимости и на самом деле она проще, чем может показаться. В действительности встроенная область видимости представляет собой всего лишь встроенный модуль под названием `builtins`, но для запрашивания встроенных имен модуль `builtins` придется импортировать, потому что имя `builtins` само по себе встроенным не является...

Нет, серьезно! Встроенная область видимости в Python 3.X реализована как стандартный библиотечный модуль по имени `builtins`, но само это имя не помещено во встроенную область видимости, так что для ее инспектирования понадобится выполнить импортацию `builtins`. Затем можно запустить вызов `dir` и посмотреть, какие имена предварительно определены. Вот как поступить в Python 3.7 (позже будет показан вариант для Python 2.X):

```
>>> import builtins
>>> dir(builtins)
['ArithmeticalError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError',
...многие другие имена не показаны...
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr',
'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
'str', 'sum', 'super',
'tuple', 'type', 'vars', 'zip']
```

Имена в полученном списке образуют встроенную область видимости в Python; примерно половина из них представляют собой встроенные исключения, а остальные – встроенные функции. Также в списке присутствуют имена `None`, `True` и `False`, хотя в Python 3.X они трактуются как зарезервированные слова. Из-за того, что Python автоматически ищет в модуле `builtins` в качестве последнего шага в своем поиске LEGB, вы получаете все имена из данного списка “даром”, т.е. можете потреблять их, не импортируя какой-либо модуль. Таким образом, на самом деле есть два способа ссылаться на встроенную функцию – использовать в своих интересах правило LEGB или вручную импортировать модуль `builtins`:

```
>>> zip                      # Обычный способ
<class 'zip'>
>>> import builtins          # Трудный путь: для настройки
>>> builtins.zip
<class 'zip'>
>>> zip is builtins.zip      # Тот же самый объект, другой поиск
True
```

Второй подход иногда удобен при решении более сложных задач, которые затрагиваются во врезке “Что потребует внимания: настройка `open`” в конце главы.

Переопределение встроенных имен: к лучшему или к худшему

Внимательный читатель мог бы также заметить, что поскольку процедура поиска LEGB захватывает *первое* вхождение имени, которое она ищет, имена в локальной области видимости могут переопределять переменные с такими же именами в глобальной и встроенной областях видимости, а глобальные имена могут переопределять

встроенные. Например, функция в состоянии создать локальную переменную по имени `open`, выполнив ее присваивание:

```
def hider():
    open = 'spam'      # Локальная переменная, скрывает здесь встроенное имя
    ...
    open('data.txt') # Ошибка: open в этой области видимости
                     # больше не открывает файл!
```

Однако созданная локальная переменная скроет встроенную функцию `open`, которая существует во встроенной (внешней) области видимости, так что имя `open` внутри данной функции больше не будет открывать файлы – теперь оно является строкой, а не функцией открытия. Это не проблема, если в функции не нужно открывать файлы, но попытка открытия файла через имя `open` приведет к ошибке.

Попасть в такую ситуацию еще проще в интерактивной подсказке, которая работает как глобальная область видимости модуля:

```
>>> open = 99          # Присваивание в глобальной области видимости
                     # здесь также скрывает встроенное имя
```

По существу нет ничего *неправильного* в том, чтобы применять встроенное имя для собственных переменных при условии, что первоначальная встроенная версия не требуется. В конце концов, если бы это было по-настоящему запрещено, то нам пришлось бы запомнить целый список встроенных имен и трактовать их как зарезервированные. Учитывая наличие свыше 150 имен внутри модуля `builtins` в Python 3.7, положение оказалось бы слишком ограничивающим и обескураживающим:

```
>>> len(dir(builtins)), len([x for x in dir(builtins) if not x.startswith('__')])
(154, 146)
```

Между прочим, при написании более сложных программ иногда возникают ситуации, когда на самом деле *желательно* заменять встроенное имя, переопределяя его в своем коде – скажем, чтобы определить специальную функцию `open`, которая контролирует попытки доступа (см. врезку “Разрушение мироздания в Python 2.X” далее в главе).

Тем не менее, переопределение встроенного имени часто является ошибкой и к тому же опасной, т.к. Python не будет выдавать предупреждение о ней. Инструменты вроде `PyChecker` (поиските информацию о нем в веб-сети) способны предупреждать о таких недоразумениях, но наилучшей защитой в данном вопросе может послужить знание: не переопределяйте встроенное имя, в котором нуждаетсясь. Если вы случайно переопределили встроенное имя в интерактивной подсказке, тогда можете либо перезапустить свой сеанс, либо выполнить оператор `del` имя и удалить переопределение из своей области видимости, тем самым восстановив оригинал из встроенной области видимости.

Обратите внимание, что функции могут аналогичным образом скрывать глобальные переменные, имена которых совпадают с именами локальных переменных. Однако поступать так полезно и фактически во многом это и есть сущность локальных областей видимости – поскольку они минимизируют возможность возникновения конфликта имен, ваши функции являются самодостаточными пространствами имен:

```
X = 88                  # Глобальная переменная X
def func():
    X = 99                # Локальная переменная X: скрывает глобальную,
                           # но именно это и желательно
func()
print(X)                # Выводит 88: значение не изменилось
```

Присваивание внутри функции создает локальную переменную X, которая полностью отличается от глобальной переменной X в модуле за пределами функции. Как следствие, изменить имя, определенное вне функции, не получится без добавления в def объявления global (или nonlocal), что объясняется в следующем разделе.



Примечание, касающееся нестыковки версий. Как ни странно, но скороговорки порой даются легче. Используемый здесь модуль builtins из Python 3.X в Python 2.X называется __builtin__. Вдобавок имя __builtins__ (с буквой s) предварительно установлено в большинстве глобальных областей видимости, включая интерактивный сеанс, для ссылки на модуль, известный как builtins в Python 3.X и __builtin__ в Python 2.X, так что часто можно применять __builtins__ без импортирования, но нельзя выполнить импортирование для самого этого имени – оно представляет собой предустановленную переменную, а не имя модуля.

То есть в Python 3.X выражение builtins is __builtins__ дает True после импортирования builtins, а в Python 2.X выражение __builtin__ is __builtins__ будет True после импортирования __builtin__. В результате мы обычно можем инспектировать встроенную область видимости, просто выполняя dir(__builtins__) без какого-либо импортирования в Python 3.X и 2.X, но рекомендуется использовать builtins для реальной работы и настройки в Python 3.X и __builtin__ для того же самого в Python 2.X. Кто сказал, что документировать такое поведение было просто?

Разрушение мироздания в Python 2.X

Есть еще одна вещь, которую вы можете делать в Python, но вероятно не должны. Из-за того, что имена True и False в Python 2.X являются всего лишь переменными во встроенной области видимости и не зарезервированы, вполне возможно присвоить их заново с помощью оператора наподобие True = False. Не волнуйтесь: поступив так, вы в действительности не нарушите логическую согласованность мироздания! Данный оператор просто переопределяет слово True для единственной области видимости, в которой он появляется, чтобы оно возвращало False. Все остальные области видимости по-прежнему ищут оригиналы во встроенной области видимости.

Однако ради еще большей забавы в Python 2.X вы могли бы выполнить __builtin__.True = False, чтобы переустановить True в False для всего процесса Python. Присваивание работает, поскольку в программе существует лишь один модуль со встроенной областью видимости, разделяемый всеми его клиентами. Увы, присваивание подобного вида в Python 3.X было запрещено, потому что True и False там трактуются как зарезервированные слова, в частности как None. Тем не менее, в Python 2.X оно переводит IDLE в странное состояние паники, сбрасывающее процесс пользовательского кода (словом, не повторяйте это дома, ребята).

Однако такая методика может быть полезной для иллюстрации лежащей в основе модели пространств имен и для разработчиков инструментов, которым требуется изменять встроенные имена вроде open с целью настройки функций. За счет переопределения имени функции во встроенной области видимости вы переустанавливаете его в настроенную версию для всех модулей в процессе. В таком случае вам, вероятно, также понадобится запомнить первоначальную версию, чтобы вызывать ее в настроенной версии – фактически один из способов создать специальную функцию open описан во врезке “Что потребует внимания: настройка open” в конце

главы после исследования замыканий вложенных областей видимости и вариантов сохранения состояния.

Кроме того, еще раз обратите внимание, что сторонние инструменты, такие как PyChecker и другие, скажем, PyLint, будут предупреждать о распространенных ошибках при программировании, в том числе о случайном присваивании встроенным именам (в инструментах подобного рода это обычно известно как “экранирование” встроенного имени). Неплохой идеей будет прогнать несколько первых программ Python через указанные инструменты, чтобы посмотреть, что они покажут.

Оператор `global`

Оператор `global` и родственный ему `nonlocal` из Python 3.X – единственные вещи в Python, которые отдаленно напоминают операторы объявления. Тем не менее, они не являются объявлениями типа или размера, а представляют собой объявления *пространства имен*. Оператор `global` сообщает Python, что функция планирует изменять одно и более глобальных имен, т.е. имен, которые существуют в области видимости (пространстве имен) включающего модуля.

Мы уже обсуждали `global`. Вот краткая сводка.

- Глобальные имена – это переменные, присвоенные на верхнем уровне включающего их файла модуля.
- Глобальные имена должны объявляться, только если им выполняются присваивания внутри функции.
- На глобальные имена можно ссылаться внутри функции без их объявления.

Другими словами, `global` дает возможность *изменять* имена, которые существуют вне `def` на верхнем уровне файла модуля. Как будет показано позже, оператор `nonlocal` почти идентичен, но применяется к именам в локальной области видимости объемлющего оператора `def`, а не к именам во включающем модуле.

Оператор `global` состоит из ключевого слова `global`, за которым следует одно и более имен, разделенных запятыми. Присваивание или ссылка на все перечисленные имена в теле функции будет приводить к их отображению на область видимости включающего модуля, например:

```
X = 88      # Глобальная переменная X
def func():
    global X
    X = 99      # Глобальная переменная X: снаружи def
func()
print(X)      # Выводит 99
```

Мы добавили в пример объявление `global`, так что имя `X` внутри `def` теперь ссылается на имя `X` снаружи `def`; на этот раз они представляют собой одну и ту же переменную, а потому изменение `X` внутри функции приводит к изменению `X` за ее пределами. Ниже показан чуть более сложный пример использования `global`:

```
y, z = 1, 2      # Глобальные переменные в модуле
def all_global():
    global x      # Объявление присваиваемой глобальной переменной
    x = y + z     # Объявлять y, z не нужно: правило LEGB
```

Переменные `x`, `y` и `z` являются глобальными в функции `all_global`. Переменные `y` и `z` глобальны из-за того, что они не присваиваются внутри функции; переменная `x` глобальна оттого, что она была указана в операторе `global` с целью явного отображения на область видимости модуля. Если бы оператор `global` отсутствовал, тогда благодаря присваиванию переменная `x` считалась бы локальной.

Обратите внимание, что переменные `y` и `z` не объявлены глобальными; Python согласно правилу поиска `LEGB` ищет их в модуле автоматически. Кроме того, до выполнения функции `all_global` переменная `x` даже не существует во включающем модуле; в данном случае первое присваивание внутри функции создает переменную `x` в модуле.

Проектирование программы: минимизируйте количество глобальных переменных

Функции в целом и глобальные переменные в частности поднимают некоторые более широкие вопросы, касающиеся проектирования. Как должны взаимодействовать наши функции? Хотя ответы на часть вопросов станут более очевидными, когда вы начнете самостоятельно писать крупные функции, несколько рекомендаций сейчас могут уберечь от возможных проблем в будущем. В общем случае функции должны полагаться на аргументы и возвращаемые значения, а не на глобальные переменные, но причины требуют пояснений.

По умолчанию имена, присваиваемые в функциях, являются локальными, поэтому если вы хотите изменять имена, существующие за пределами функций, то вам придется писать дополнительный код (например, операторы `global`). Так было задумано – как принято в Python, чтобы сделать что-то потенциально “неправильное”, необходимо написать добавочный код. Несмотря на то что встречаются ситуации, когда глобальные переменные удобны, присваиваемые внутри `def` переменные по умолчанию будут локальными, поскольку так предусмотрено наилучшей стратегией. Изменение глобальных переменных может привести к общезвестным проблемам при разработке программного обеспечения: из-за того, что значения переменных зависят от порядка вызовов произвольно отдаленных функций, программы могут стать трудными для отладки и вообще для восприятия.

Возьмем в качестве примера следующий файл модуля, который импортируется и применяется где-то в другом месте:

```
X = 99
def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```

Предположим, что необходимо модифицировать или повторно использовать данный код. Каким будет значение `X`? По правде говоря, вопрос не имеет смысла, если только не привязать его ко времени – значение `X` обусловлено выбранным моментом времени, т.к. оно зависит от того, которая из функций вызывалась последней (чего мы не можем определить по приведенному одному файлу).

Совокупный эффект заключается в том, что для понимания кода вам понадобится отследить поток управления через *целую программу*. И если необходимо повторно применить или модифицировать код, тогда вам придется держать в уме всю программу

целиком. В таком случае вы не сможете использовать одну из функций без привлечения другой. Они зависят от глобальной переменной – т.е. *связаны с ней*. Именно так выглядит проблема с глобальными переменными: они обычно делают код более сложным для понимания и многократного применения, чем код, состоящий из самодостаточных функций, которые полагаются на локальные переменные.

С другой стороны, исключая инструменты вроде замыканий вложенных областей видимости или объектно-ориентированное программирование с помощью классов, глобальные переменные в Python предлагают наиболее прямолинейный способ сохранения разделяемой *информации о состоянии* – информации, которую функция должна запоминать для использования при вызове в следующий раз. После возвращения потока управления из функции локальные переменные исчезают, но глобальные переменные остаются. Как вы увидите позже, такой цели можно также достичь посредством других методик, которые допускают наличие множества копий сохраненной информации, но в целом они гораздо сложнее помещения значений в глобальную область видимости для сохранения в простых ситуациях, когда данный подход применим.

Более того, некоторые программы назначают отдельный модуль для сбора в нем глобальных переменных; так как это ожидается, оно не настолько вредно. Программы, использующие многопоточность для организации параллельной обработки в Python, также обычно полагаются на глобальные переменные – они становятся разделяемой памятью между функциями, выполняющимися в параллельных потоках, и потому действуют как коммуникационный механизм².

Однако на данный момент, особенно если вы только начинаете изучать программирование, старайтесь избегать применения глобальных переменных везде, где только можно – они затрудняют понимание и многократное использование программ, а также не работают в ситуациях, когда одной копии сохраненных данных недостаточно. Попробуйте взамен обмениваться данными с помощью передаваемых аргументов и возвращаемых значений. Пройдет полгода, и вы с коллегами будете счастливы, что поступили так.

Проектирование программы: минимизируйте количество межфайловых изменений

Существует еще одна проблема проектирования, касающаяся областей видимости: хотя мы можем изменять переменные в другом файле напрямую, обычно мы не должны это делать. Файлы модулей были представлены в главе 3 и более глубоко раскрываются в следующей части книги. В качестве иллюстрации их взаимосвязи с областями видимости рассмотрим приведенные ниже два файла модулей:

```
# first.py  
X = 99           # Этому коду ничего не известно о second.py
```

² Многопоточность обеспечивает выполнение вызовов функций параллельно с остальной программой и поддерживается в Python стандартными библиотечными модулями `_thread`, `threading` и `queue` (`thread`, `threading` и `Queue` в Python 2.X). Поскольку все потоковые функции выполняются в том же самом процессе, глобальные области видимости часто служат одной из форм разделяемой памяти между ними (потоки могут разделять имена в глобальных областях видимости, а также объекты в пространстве памяти процесса). Многопоточность обычно используется для длительно выполняющихся задач в графических пользовательских интерфейсах, чтобы реализовать неблокирующие операции и довести до максимума производительность центрального процессора. Многопоточность выходит за рамки тем, рассматриваемых в книге; за более подробными сведениями обращайтесь к руководству по библиотеке Python и материалам, перечисленным в предисловии (таким как книга *Programming Python* изданная O'Reilly (<http://shop.oreilly.com/product/9780596158118.do>)).

```
# second.py
import first
print(first.X)      # Нормально: ссылка на имя из другого файла
first.X = 88        # Но его изменение может быть слишком тонким и неявным
```

В первом модуле определяется переменная X, которая во втором модуле вводится и затем изменяется путем присваивания. Обратите внимание, что во втором модуле мы должны сначала импортировать первый модуль, чтобы вообще получить доступ к его переменной – как уже известно, каждый модуль является самодостаточным пространством имен (пакетом переменных), и мы обязаны импортировать один модуль, чтобы видеть его содержимое в другом. В том и заключается сущность модулей: за счет разделения переменных на пофайловой основе удается избежать конфликтов имен между файлами во многом подобно тому, как локальные переменные позволяют не допускать конфликтов имен среди функций.

Тем не менее, с точки зрения тематики настоящей главы глобальная область видимости файла модуля на самом деле *становится* пространством имен атрибутов объекта модуля после его импортирования. Импортирующие модули автоматически получают доступ ко всем глобальным переменным файла, потому что глобальная область видимости модуля превращается в пространство имен атрибутов объекта, когда модуль импортируется.

После импортирования первого модуля второй модуль выполняет вывод его переменной и затем присваивает ей новое значение. Ссылаясь на переменную модуля для ее вывода вполне нормально – именно так модули увязываются вместе в более крупную систему. Однако проблема с присваиванием first.X заключается в том, что оно чрезмерно скрыто: разработчику, отвечающему за сопровождение или многократное применение первого модуля, вряд ли известно о том, что какой-то произвольно далеко стоящий в цепочке наследования модуль может изменять X во время выполнения. В действительности второй модуль может находиться в совершенно другом каталоге, так что его вообще будет сложно заметить.

Хотя межфайловые изменения переменных вполне возможны в Python, они обычно гораздо более неуловимые, нежели бы вам хотелось. Вдобавок такие изменения делают *связность* двух файлов слишком тесной – поскольку они оба зависят от значения переменной X, понять или повторно использовать один файл без другого затруднительно. Такие неявные межфайловые зависимости могут в лучшем случае привести к негибкому коду, в худшем – к откровенным ошибкам.

И снова рекомендуется в целом не поступать так – наилучший способ взаимодействия между границами файлов предусматривает вызов функций с передачей им аргументов и получением обратно возвращаемых значений. В этом особом случае, возможно, было бы разумно написать *функцию доступа* для управления изменением:

```
# first.py
X = 99

def setX(new):      # Функция доступа делает внешние изменения явными
    global X        # И она позволяет управлять доступом в одном месте
    X = new

# second.py
import first
first.setX(88)     # Вызывать функцию вместо изменения напрямую
```

Такой подход требует большего объема кода и может показаться тривиальным изменением, но он демонстрирует огромное отличие в плане читабельности и удобства сопровождения – когда разработчик, читающий только код первого модуля, увидит

функцию, он будет знать, что она входит в состав *интерфейса*, и ожидать изменения X. Другими словами, функция доступа устраниет элемент неожиданности, который редко считается чем-то хорошим в программных проектах. Несмотря на то что мы не в состоянии вообще предотвратить межфайловые изменения, здравый смысл подсказывает, что они должны быть сведены к минимуму, если только не являются широко принятыми в программе.



Когда мы встретимся с классами в части VI, то увидим похожие методики для написания кода средств доступа к атрибутам. В отличие от модулей классы также способны автоматически перехватывать извлечения атрибутов с помощью перегрузки операций, даже если их клиенты не применяют средства доступа.

Другие способы доступа к глобальным переменным

Интересно отметить, что по причине превращения переменных из глобальной области видимости в атрибуты объекта загруженного модуля мы можем эмулировать оператор `global`, импортируя включающий модуль и выполняя присваивание его атрибутам, как показано в следующем примере файла модуля. Код в этом файле импортирует включающий модуль сначала по имени, а затем путем индексации таблицы загруженных модулей `sys.modules` (которая обсуждается в главах 22 и 25):

```
# thismod.py
var = 99                                     # Глобальная переменная == атрибут модуля
def local():
    var = 0                                    # Изменение локальной переменной
def globl():
    global var                                # Объявление глобальной переменной
    var += 1                                  # (нормальное)
                                                # Изменение глобальной переменной
def glob2():
    var = 0                                    # Изменение локальной переменной
    import thismod                            # Импортирование самого себя
    thismod.var += 1                          # Изменение глобальной переменной
def glob3():
    var = 0                                    # Изменение локальной переменной
    import sys                                 # Импортирование системной таблицы
    glob = sys.modules['thismod']             # Получение объекта модуля
    glob.var += 1                            # (либо использовать __name__)
                                            # Изменение глобальной переменной
def test():
    print(var)
    local(); globl(); glob2(); glob3()
    print(var)
```

Выполнение приводит к добавлению 3 к глобальной переменной (на нее не оказывает влияния только первая функция):

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

Код иллюстрирует эквивалентность глобальных переменных и атрибутов модуля, но требует гораздо большего объема работы по сравнению с использованием оператора `global` для пояснения своих намерений.

Как мы видели, оператор `global` позволяет изменять имена из модуля, внешнего по отношению к функции. Он имеет близкородственный оператор по имени `nonlocal`, который можно применять для изменения имен из объемлющих функций, но прежде чем выяснить, какую пользу из этого можно извлечь, необходимо исследовать объемлющие функции вообще.

Области видимости и вложенные функции

До сих пор я нарочно опустил одну часть правил, касающихся областей видимости Python, потому что она относительно редко встречается на практике. Тем не менее, пришло время внимательнее посмотреть на букву *E* в правиле поиска LEGB. Уровень *E* был добавлен в версии Python 2.2; он принимает форму локальных областей видимости любых объемлющих функций. Объемлющие области видимости временами также называются *статически вложенными областями видимости*. Вообще говоря, вложение является лексическим — вложенные области видимости соответствуют физически и синтаксически вложенным кодовым структурам в исходном тексте программы.

Детали вложенных областей видимости

С добавлением вложенных областей видимости слегка усложняются правила поиска переменных. Вот что происходит внутри функции.

- Ссылка (*X*) ищет имя *X* сначала в текущей локальной области видимости (функция); затем в локальных областях видимости любых лексически объемлющих функций в исходном коде, от внутренней до наружной; далее в текущей глобальной области видимости (файл модуля); и, наконец, во встроенной области видимости (модуль `builtins`). Объявления `global` заставляют поиск взамен начинаться в глобальной области видимости (файл модуля).
- Присваивание (*X = значение*) по умолчанию создает либо изменяет имя *X* в текущей локальной области видимости. Если имя *X* объявлено глобальным внутри функции, тогда присваивание создает или изменяет имя *X* в области видимости включающего модуля. Если же имя *X* объявлено нелокальным внутри функции в Python 3.X (только), то присваивание изменяет имя *X* в локальной области видимости ближайшей объемлющей функции.

Обратите внимание, что объявление `global` по-прежнему отображает переменные на область видимости включающего модуля. Когда присутствуют вложенные функции, на переменные из объемлющих функций можно ссылаться, но для их изменения в Python 3.X требуются объявления `nonlocal`.

Примеры вложенных областей видимости

Чтобы прояснить моменты, указанные в предыдущем разделе, давайте обратимся к реальному коду. Вот на что похожа область видимости объемлющей функции (для запуска кода поместите его в файл сценария или наберите в интерактивной подсказке):

```
X = 99          # Имя в глобальной области видимости: не используется
def f1():
    X = 88      # Локальное имя объемлющего def
```

```
def f2():
    print(X)      # Ссылка во вложенном def
f2()
f1()                  # Выводит 88: локальное имя объемлющего def
```

Первым делом это законный код Python: `def` представляет собой просто исполняемый оператор, который может появляться везде, где допускается любой другой оператор — в том числе быть вложенным в другой `def`. Здесь вложенный оператор `def` запускается, пока выполняется функция `f1`; он создает объект функции и присваивает его имени `f2`, т.е. локальной переменной внутри локальной области видимости `f1`. В определенном смысле `f2` представляет собой временную функцию, которая существует только в период выполнения (и видима только в коде) объемлющей функции `f1`.

Но взгляните, что происходит внутри функции `f2`: когда она выводит переменную `X`, то ссылается на `X`, которая существует в локальной области видимости объемлющей функции `f1`. Поскольку функции могут получать доступ к именам во всех физически объемлющих операторах `def`, согласно правилу поиска LEGB имя `X` в `f2` автоматически отображается на имя `X` в `f1`.

Поиск в объемлющей области видимости работает, даже если уже произошел возврат из объемлющей функции. Скажем, в следующем коде определена функция, которая создает и возвращает объект другой функции, представляя более распространенный шаблон использования:

```
def f1():
    X = 88
def f2():
    print(X)      # Помнит значение X из области видимости объемлющего def
    return f2      # Возвращает объект функции f2, но не вызывает функцию
action = f1()      # Создает и возвращает объект функции
action()           # Вызов функции: выводит 88
```

Вызов `action` на самом деле выполняет функцию по имени `f2`, которая была создана во время выполнения `f1`. Прием работает, потому что подобно всему остальному функции в Python являются объектами и могут передаваться как возвращаемые значения из других функций. Что еще более важно, `f2` помнит значение `X` из объемлющей области видимости функции `f1`, хотя `f1` больше не активна — это подводит нас к следующей теме.

Фабричные функции: замыкания

В зависимости от того, у кого вы спросите, поведение такого вида называют *замыканием* или *фабричной функцией* — первый термин описывает методику *функционального программирования*, а второй обозначает *паттерн проектирования*. Как бы он ни назывался, указанный объект функции помнит значения из объемлющих областей видимости, невзирая на то, присутствуют ли еще эти области видимости в памяти. Фактически он имеет присоединенные пакеты данных из памяти (известные как *сохранение состояния*), которые локальны для каждой созданной копии вложенной функции и часто выступают в качестве простой альтернативы классам в такой роли.

Простая фабричная функция

Фабричные функции (они же замыкания) иногда применяются в программах, которым необходимо генерировать обработчики событий на лету в ответ на условия, сложившиеся во время выполнения. Например, представим себе графический пользо-

вательский интерфейс, где нужно определять действия согласно данным, введенным пользователем, которые невозможно предугадать на этапе построения интерфейса. В таких ситуациях нам требуется функция, создающая и возвращающая другую функцию с информацией, которая может варьироваться в зависимости от создаваемой функции.

Чтобы продемонстрировать прием на простом примере, рассмотрим следующую функцию, набранную в интерактивной подсказке (и приведенную здесь без приглашений к продолжению . . ., как описано далее в замечаниях по представлению):

```
>>> def maker(N):
    def action(X):      # Создание и возвращение функции action
        return X ** N   # action сохраняет N из объемлющей области видимости
    return action
```

В коде определяется внешняя функция, которая просто генерирует и возвращает вложенную функцию, не вызывая ее — `maker` создает `action`, но лишь возвращает `action` без выполнения. Если мы вызовем внешнюю функцию:

```
>>> f = maker(2)    # Передача 2 аргументу N
>>> f
<function maker.<locals>.action at 0x0000000002A4A158>
```

то получим обратно ссылку на сгенерированную вложенную функцию — ту, что была создана при выполнении вложенного оператора `def`. Вызов результата, возвращенного внешней функцией:

```
>>> f(3)      # Передача 3 аргументу X, в N запоминается 2: 3 ** 2
9
>>> f(4)      # 4 ** 2
16
```

приводит к запуску вложенной функции, названной `action` внутри `maker`. Другими словами, мы вызываем вложенную функцию, которую создала и возвратила функция `maker`.

Пожалуй, самой необычной частью следует считать *запоминание вложенной функцией целого числа 2*, т.е. значения переменной `N` в `maker`, хотя к моменту вызова `action` уже произошел возврат из функции `maker`, и она закончила работу. По существу `N` из локальной области видимости объемлющей функции сохраняется как информация о состоянии, присоединенная к сгенерированной функции `action` — вот почему в результате ее вызова мы получили значение аргумента, возведенное в квадрат.

Не менее важен и тот факт, что если мы снова вызовем внешнюю функцию, то получим обратно *новую* вложенную функцию с *другой* присоединенной информацией о состоянии. Так, при вызове новой функции значение аргумента возводится в куб, а не в квадрат, но предыдущая функция по-прежнему возводит в квадрат, как и ранее:

```
>>> g = maker(3)      # g запоминает 3, f запоминает 2
>>> g(4)              # 4 ** 3
64
>>> f(4)              # 4 ** 2
16
```

Подобное возможно из-за того, что каждый вызов фабричной функции получает *собственный* набор информации о состоянии. В нашем случае функция, присвоенная имени `g`, запоминает 3, а `f` запоминает 2, т.к. каждая имеет свою информацию о состоянии, хранимую в переменной `N` из `maker`.

Это довольно сложный прием, который не особенно часто встречается в большей части кода, и он может быть популярен среди программистов с опытом использования языков функционального программирования. С другой стороны, объемлющие области видимости нередко употребляются выражениями создания функций `lambda`, которые объясняются далее — являясь выражениями, они почти всегда вкладываются внутрь `def`. Скажем, в нашем примере выражение `lambda` могло бы заменить оператор `def`:

```
>>> def maker(N):
    return lambda X: X ** N      # Функции lambda тоже сохраняют состояние
>>> h = maker(3)
>>> h(4)                      # Снова 4 ** 3
64
```

За более реалистичным примером замыканий в работе обращайтесь к врезке “Что потребует внимания: настройка `open`” в конце главы. Там применяются похожие методики для сохранения информации с целью дальнейшего использования в объемлющей области видимости.



Замечания по представлению. В этой главе я начал приводить примеры интерактивного взаимодействия без *приглашений к продолжению* . . ., которые в вашем интерфейсе могут отображаться или не отображаться (они отображаются в интерактивной подсказке, но не в IDLE). Такое соглашение будет соблюдаться и далее для крупных примеров кода. Я полагаю, что к настоящему времени вы уже хорошо понимаете правила применения отступов и обрели достаточный опыт в наборе кода Python. Кроме того, я буду представлять все больше и больше кода отдельно или в файлах, произвольно переключаясь между ними и интерактивным вводом.

Замыкания или классы, раунд 1

Классы, подробно описываемые в части VI книги, некоторым могут показаться лучшим вариантом для такого хранения информации о состоянии, поскольку они обеспечивают более явное запоминание посредством присваивания атрибутов. Классы также напрямую поддерживают дополнительные инструменты, которые отсутствуют в функциях замыканий, вроде настройки путем наследования и перегрузки операций, и более естественно реализуют множество линий поведения в форме методов. По причине таких отличий классы могут быть эффективнее при реализации усовершенствованных объектов.

Однако если сохранение состояния является единственной целью, тогда функции замыканий часто оказываются более легковесной и жизнеспособной альтернативой. Они предоставляют каждому вызову локализованное хранилище для данных, требующихся одиночной вложенной функции. Это особенно верно, когда мы добавляем рассматриваемый позже оператор `nonlocal` из Python 3.X, чтобы разрешить изменять состояние в объемлющей области видимости (в Python 2.X объемлющие области видимости допускают только чтение, а потому с ними связаны ограниченные сценарии использования).

В более широком смысле функциям Python доступны многие способы сохранения состояния между вызовами. Несмотря на то что значения нормальных локальных переменных исчезают, когда управление возвращается из функций, значения могут сохраняться между вызовами в глобальных переменных, в атрибутах экземпляров классов, в обсуждаемых здесь ссылках из объемлющих областей видимости, а также в

стандартных значениях аргументов и атрибутах функций. Некоторые могут включить в данный список и изменяемые стандартные аргументы (хотя другие могут пожелать, чтобы это не делалось).

Мы кратко обсудим альтернативы и коснемся атрибутов функций позже в главе, а полную историю об аргументах и стандартных значениях представим в главе 18. Тем не менее, в следующем разделе предлагается введение, которое поможет оценить, каким образом стандартные значения участвуют в сохранении состояния.



Замыкания также могут создаваться, когда определение `class` вкладываеться в оператор `def`: значения локальных имен объемлющей функции сохраняются за счет ссылок внутри класса или одного из его функций методов. Дополнительные сведения о вложенных классах ищите в главе 29. Как вы увидите в примерах далее в книге (скажем, при обсуждении декораторов в главе 39), внешний оператор `def` в таком коде исполняет похожую роль: он становится фабрикой классов и обеспечивает сохранение состояния для вложенного класса.

Сохранение состояния из объемлющей области видимости с помощью стандартных значений

В ранних версиях Python (до Python 2.2) разновидность кода из предыдущего раздела потерпела бы неудачу, потому что вложенные операторы `def` ничего не делали в отношении областей видимости. В показанном ниже коде ссылка на переменную внутри `f2` инициировала бы поиск только в локальной (`f2`), далее в глобальной (код вне `f1`) и затем во встроенной области видимости. Из-за того, что поиск пропускал области видимости объемлющих функций, результатом была ошибка. В качестве обходного приема программисты обычно применяли *стандартные значения аргументов* для передачи и запоминания объектов в объемлющей области видимости:

```
def f1():
    x = 88
    def f2(x=x):
        # Запоминает X из объемлющей области видимости
        # посредством стандартных значений
        print(x)
    f2()
f1()                                # Выводит 88
```

Такой стиль написания кода подходит для всех выпусков Python и вы будете по-прежнему встречать данный шаблон в существующем коде Python. На самом деле, как вскоре будет показано, он все еще *обязателен* для переменных цикла и потому заслуживает изучения даже в наши дни. Если кратко, то синтаксис `arg=val` в заголовке `def` означает, что аргумент `arg` по умолчанию получит значение `val`, когда никакого реального значения для `arg` в вызове не передается. Здесь этот синтаксис используется для явной установки подлежащего сохранению состояния из объемлющей области видимости.

В частности, в модифицированной функции `f2`, показанной выше, `x=x` означает, что аргумент `x` по умолчанию получит значение переменной `x` из объемлющей области видимости — поскольку аргумент `x` оценивается до того, как Python переходит внутрь вложенного оператора `def`, он по-прежнему ссылается на `x` в `f1`. В сущности, стандартное значение аргумента запоминает, каким было значение `x` в `f1`: объектом 88.

Все это довольно-таки сложно и полностью зависит от того, когда происходит оценка стандартного значения. В действительности правило поиска во вложенных областях видимости было добавлено в Python для того, чтобы сделать стандартные

значения ненужными в данной роли – в наше время Python автоматически запоминает любые значения в объемлющей области видимости, которые требуется применять во вложенных операторах `def`.

Разумеется, наилучшая рекомендация для большей части кода заключается в том, чтобы избегать вложения операторов `def` внутрь `def`, т.к. тогда программа станет гораздо проще – согласно духу Python плоский код, как правило, лучше вложенного. Ниже приведен эквивалент предыдущего примера, в котором вложение вообще отсутствует. Обратите внимание на опережающую ссылку в коде – вполне нормально вызывать функцию, определенную после функции, где она вызывается, при условии, что второй оператор `def` выполняется перед фактическим вызовом первой функции. Код внутри `def` никогда не выполняется до тех пор, пока функция не будет действительно вызвана:

```
>>> def f1():
    x = 88          # Передача x вместо вложения
    f2(x)          # Опережающая ссылка допустима

>>> def f2(x):
    print(x)      # Плоский код все еще нередко лучше вложенного!

>>> f1()
88
```

Если вы избегаете вложения подобным способом, то можете в основном забыть о концепции вложенных областей видимости в Python. С другой стороны, вложенные функции фабричных функций (замыканий) довольно распространены в современном коде Python как функции `lambda`, которые почти естественно встречаются вложеными в операторы `def` и часто полагаются на уровень вложенных областей видимости, что будет обсуждаться в следующем разделе.

Вложенные области видимости, стандартные значения и выражения `lambda`

Несмотря на растущее использование в операторах `def` в наше время, наиболее вероятно, что вас будут заботить области видимости вложенных функций, когда вы начнете писать код или читать выражения `lambda`. Мы уже бегло встречались с конструкцией `lambda` ранее и подробно опишем ее в главе 19, а пока отметим, что это выражение, которое генерирует новую функцию для вызова в будущем, почти как оператор `def`. Однако будучи выражением, `lambda` может применяться в местах, где оператор `def` не допускается, скажем, внутри списковых и словарных литералов.

Подобно `def` выражение `lambda` также вводит новую локальную область видимости для функции, которую создает. Благодаря уровню поиска в объемлющих областях видимости выражения `lambda` могут видеть все переменные, существующие в функциях, в которых эти выражения записаны. Таким образом, следующий код – вариация ранее показанной фабрики – работает, но только по причине применения правил поиска во вложенных областях видимости:

```
def func():
    x = 4
    action = (lambda n: x ** n)           # x запоминается из объемлющего def
    return action

x = func()
print(x(2))                            # Выводит 16, 4 ** 2
```

До того, как появились области видимости вложенных функций, для передачи значений из объемлющей области видимости в выражения `lambda` программисты

использовали стандартные значения в точности как для операторов `def`. Например, приведенный ниже код работает во всех выпусках Python:

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n)      # Передача x вручную
    return action
```

Поскольку `lambda` являются выражениями, они естественно (и даже обычно) вкладываются внутрь объемлющих операторов `def`. Следовательно, видимо именно они извлекли наибольшую выгоду от добавления в правила поиска областей видимости вложенных функций; в большинстве случаев теперь отсутствует необходимость передавать значения в выражения `lambda` с помощью стандартных значений.

Переменные цикла могут требовать стандартные значения, а не области видимости

Есть одно заметное исключение из только что изложенного правила (и причина, по которой была показана вышедшая из употребления методика со стандартными значениями аргументов). Если выражение `lambda` или оператор `def`, определяемый внутри функции, вкладывается в цикл, а вложенная функция ссылается на переменную из объемлющей области видимости, которая изменяется этим циклом, то все функции, генерируемые внутри цикла, будут иметь одно и то же значение – значение ссылаемой переменной на *последней* итерации цикла. В таких случаях для сохранения *текущего* значения переменной приходится применять стандартные значения.

Ситуация может показаться не вполне ясной, но на практике она встречается чаще, чем вы могли подумать, особенно в коде, который генерирует функции обработчиков обратного вызова для виджетов в графическом пользовательском интерфейсе – например, обработчики щелчков для всех кнопок в строке. Если они создаются в цикле, тогда может возникнуть необходимость в сохранении состояния посредством стандартных значений, иначе обратные вызовы всех кнопок в итоге станут выполнять одно и то же действие.

Ниже представлена упрощенная иллюстрация данного явления: здесь предпринимается попытка построить список функций, каждая из которых запоминает текущее значение переменной `i` из объемлющей области видимости:

```
>>> def makeActions():
    acts = []
    for i in range(5):                  # Попытка запомнить каждое значение i
        acts.append(lambda x: i ** x)   # Но все функции запоминают
                                         # последнее значение i!
    return acts

>>> acts = makeActions()
>>> acts[0]
<function makeActions.<locals>.<lambda> at 0x0000000002A4A400>
```

Тем не менее, код не работает должным образом – из-за того, что переменная из объемлющей области видимости ищется, когда функции *вызываются* в более позднее время, все они фактически запоминают то же самое значение: значение, которое переменная цикла имела на *последней* итерации цикла. То есть при передаче аргумента степень 2 в каждом из следующих вызовов мы получаем обратно 4, возведенное в степень 2, для каждой функции в списке, т.к. значение `i` во всех функциях одинаково – 4:

```
>>> acts[0](2)                      # Все возвращают 4 ** 2; 4 - последнее значение i
```

```
16
>>> acts[1](2)      # Здесь должно быть 1 ** 2 (1)
16
>>> acts[2](2)      # Здесь должно быть 2 ** 2 (4)
16
>>> acts[4](2)      # И только здесь должно быть 4 ** 2 (16)
16
```

Мы столкнулись с ситуацией, когда по-прежнему приходится явно сохранять значения переменной из объемлющей области видимости с помощью стандартных значений аргументов, а не ссылок на переменную в объемлющей области видимости. Таким образом, чтобы обеспечить работу подобного кода, мы должны передавать *текущее* значение переменной из объемлющей области видимости посредством стандартного значения. Поскольку стандартные значения оцениваются при *создании* вложенных функций (а не их *вызове* в более позднее время), каждая запоминает собственное значение *i*:

```
>>> def makeActions():
    acts = []
    for i in range(5):           # Использование стандартных значений
        acts.append(lambda x, i=i: i ** x) # Запоминает текущее значение i
    return acts

>>> acts = makeActions()
>>> acts[0](2)                # 0 ** 2
0
>>> acts[1](2)                # 1 ** 2
1
>>> acts[2](2)                # 2 ** 2
4
>>> acts[4](2)                # 4 ** 2
16
```

Подход может выглядеть как предрасположенный к изменениям артефакт реализации, важность которого возрастает при написании более крупных программ. Стандартные значения более подробно обсуждаются в главе 18, а *lambda* – в главе 19; после чтения указанных глав можете еще раз пересмотреть текущий раздел³.

Произвольное вложение областей видимости

Прежде чем закончить обсуждение, следует отметить, что области видимости могут вкладываться произвольно глубоко, но при ссылке на имена их поиск производится только в операторах *def* объемлющих функций (не классов, описанных в части VI):

```
>>> def f1():
    x = 99
```

³ В разделе “Затруднения, связанные с функциями” главы 21 мы также увидим, что существует похожая проблема с использованием изменяемых объектов вроде списков и словарей для стандартных значений аргументов (например, *def f(a=[])*) – поскольку стандартные значения реализованы как одиночные объекты, присоединенные к функциям, изменяемые объекты в качестве стандартных сохраняют состояние от вызова к вызову, а не инициализируются заново при каждом вызове. В зависимости от того, у кого вы спросите, это считается либо средством, поддерживающим еще один способ реализации хранения состояния, либо странной особенностью языка; более подробно данный аспект обсуждается в конце главы 21.

```
def f2():
    def f3():
        print(x)          # Находится в локальной области видимости f1!
    f3()
f2()
>>> f1()
99
```

Python будет искать в локальных областях видимости *всех* объемлющих `def`, изнутри наружу, выше локальной области видимости ссылающейся функции и ниже глобальной области видимости модуля или встроенных областей видимости. Однако на практике код такого рода встречается еще реже. И мы снова говорим, что в Python *плоский код лучше вложенного*, и это остается в целом верным даже с добавлением замыканий вложенных областей видимости. За исключением ограниченных контекстов ваша жизнь (и жизнь коллег) обычно становится проще, если вы будете сводить к минимуму вложенные определения функций.

Оператор `nonlocal` в Python 3.X

В предыдущем разделе мы исследовали способ, которым вложенные функции могут *ссылаться* на переменные в области видимости объемлющей функции, даже если объемлющая функция уже произвела возврат. Оказывается, что в Python 3.X (хотя не в Python 2.X) мы также можем *изменять* такие переменные из объемлющей области видимости при условии их объявления в операторах `nonlocal`. Благодаря оператору `nonlocal` вложенные определения `def` могут иметь доступ по чтению и записи к именам в объемлющих функциях, что делает замыкания вложенных областей видимости более полезными, предоставляя изменяемую информацию о состоянии.

Оператор `nonlocal` по своей форме и роли похож на рассмотренный ранее оператор `global`. Подобно `global` оператор `nonlocal` заявляет, что будет изменяться имя из объемлющей области видимости. Тем не менее, в отличие от `global` оператор `nonlocal` применяется к имени в области видимости объемлющей функции, а не глобальной области видимости модуля за пределами всех `def`. Также в отличие от `global` при объявлении имен в `nonlocal` они уже должны существовать в области видимости объемлющей функции — имена могут существовать только в объемлющих функциях и не могут создаваться первым присваиванием во вложенном операторе `def`.

Другими словами оператор `nonlocal` делает возможным присваивание значений именам из областей видимости объемлющих функций и ограничивает поиск таких имен областями видимости объемлющих `def`. Совокупным эффектом является более прямая и надежная реализация изменяемой информации о состоянии в контекстах, для которых использовать классы с атрибутами, наследованием и множеством линий поведения нежелательно или нет необходимости.

Основы оператора `nonlocal`

В Python 3.X появился новый оператор `nonlocal`, имеющий смысл только внутри функций:

```
def func():
    nonlocal name1, name2, ...    # Здесь нормально
>>> nonlocal x
SyntaxError: nonlocal declaration not allowed at module level
Синтаксическая ошибка: объявление nonlocal не разрешено на уровне модуля
```

Оператор `nonlocal` позволяет вложенной функции изменять одно или большее количество имен, определенных в области видимости синтаксически объемлющей функции. В Python 2.X, когда один оператор `def` функции вложен в другой, то вложенная функция способна ссылаться на любые имена, определенные присваиваниями в области видимости объемлющего оператора `def`, но не может изменять их. В Python 3.X объявление имен из объемлющих областей видимости в операторе `nonlocal` дает возможность вложенным функциям выполнять присваивание и тем самым изменять такие имена.

В итоге объемлющие функции располагают способом предоставления *записывающей* информации о состоянии, запоминаемой при вызове вложенной функции в более позднее время. Разрешение состоянию изменяться делает его более полезным для вложенной функции (в качестве примера представьте себе счетчик в объемлющей области видимости). В Python 2.X программисты обычно достигали похожих целей за счет использования классов или других схем. Однако из-за того, что вложенные функции стали более распространенным кодовым шаблоном для реализации хранения состояния, оператор `nonlocal` делает их универсально применимыми.

Помимо разрешения именам из объемлющих `def` изменяться оператор `nonlocal` также навязывает ограничение для ссылок – во многом подобно `global` оператор `nonlocal` приводит к тому, что поиск перечисленных в нем имен начинается в областях видимости объемлющих `def`, а не в локальной области видимости объявляющей функции. То есть `nonlocal` также означает “полностью пропустить мою локальную область видимости”.

В действительности имена, перечисленные в операторе `nonlocal`, уже должны быть определены в объемлющем операторе `def`, когда достигается `nonlocal`, в противном случае возникает ошибка. Конечный эффект очень похож на `global`: оператор `global` означает, что имена находятся во включающем модуле, а `nonlocal` – что они пребывают в объемлющем `def`. Оператор `nonlocal` даже более строгий – поиск в областях видимости ограничивается *только* объемлющими `def`. Таким образом, нелокальные имена могут появляться только в объемлющих `def`, но не в глобальной области видимости модуля или встроенной области видимости за пределами операторов `def`.

Добавление `nonlocal` в целом не изменяет правила поиска ссылок на имена в областях видимости; они продолжают работать прежним образом согласно описанному ранее правилу LEGB. Оператор `nonlocal` служит по большей части для того, чтобы позволить именам из объемлющих областей видимости изменяться, а не просто ссылаться на них. Тем не менее, операторы `global` и `nonlocal` ужесточают и даже в чем-то ограничивают правила поиска, когда они размещены внутри функции.

- Оператор `global` заставляет поиск в областях видимости начинаться в области видимости включающего модуля и открывает возможность присваивания находящимся там именам. Если имя не существует в модуле, тогда поиск продолжается во встроенной области видимости, но присваивания глобальным именам всегда создают или изменяют их в области видимости модуля.
- Оператор `nonlocal` ограничивает поиск в областях видимости только объемлющими `def`, требует, чтобы имена уже там существовали, и делает возможным присваивание им значений. Поиск не продолжается в глобальной или встроенной областях видимости.

В Python 2.X разрешена ссылка на имена из областей видимости объемлющих `def`, но не присваивание. Однако для получения того же эффекта изменяемой информации о состоянии, что и оператор `nonlocal`, вы по-прежнему можете использовать классы с явными атрибутами (а в ряде контекстов поступать так может быть даже лучше);

временами достичь похожих целей удается также с помощью глобальных имен и атрибутов функций. Вскоре мы обсудим это более подробно, но сначала давайте взглянем на рабочий код.

Оператор `nonlocal` в действии

Все приведенные примеры выполняются в Python 3.X. Ссылки на имена из областей видимости объемлющих `def` работают в Python 3.X как и в Python 2.X – в следующем коде `tester` создает и возвращает функцию `nested`, подлежащую вызову в более позднее время, а ссылка на `state` в `nested` отображается на имя в локальной области видимости `tester` с применением обычных правил поиска в областях видимости:

```
C:\code> c:\python33\python
>>> def tester(start):
    state = start      # Ссылка на нелокальные переменные работает нормально
    def nested(label):
        print(label, state) # Запоминает state из объемлющей области видимости
        return nested
    >>> F = tester(0)
    >>> F('spam')
spam 0
>>> F('ham')
ham 0
```

Тем не менее, изменение имени из области видимости объемлющего `def` по умолчанию не разрешено; это также нормальное поведение в Python 2.X:

```
>>> def tester(start):
    state = start
    def nested(label):
        print(label, state)
        state += 1 # По умолчанию изменять нельзя (в Python 2.X вообще никогда)
    return nested
    >>> F = tester(0)
    >>> F('spam')
UnboundLocalError: local variable 'state' referenced before assignment
Ошибка несвязанной локальной переменной: ссылка на локальную переменную
state перед ее присваиванием
```

Использование оператора `nonlocal` для изменений

Если теперь в Python 3.X объявить переменную `state` из области видимости `tester` как `nonlocal` внутри `nested`, то мы сможем ее также изменять во вложенной функции. Прием работает, несмотря на то, что к моменту вызова возвращенной функции `nested` через имя `F` функция `tester` уже завершилась:

```
>>> def tester(start):
    state = start      # Каждый вызов получает собственное значение state
    def nested(label):
        nonlocal state # Запоминает из объемлющей области видимости
        print(label, state)
        state += 1      # Нелокальную переменную разрешено изменять
    return nested
    >>> F = tester(0)
    >>> F('spam')      # При каждом вызове state инкрементируется
```

```
spam 0
>>> F('ham')
ham 1
>>> F('eggs')
eggs 2
```

Как обычно со ссылками на переменные из объемлющих областей видимости, мы можем вызывать фабричную функцию (замыкание) `tester` много раз для получения множества копий ее состояния в памяти. Объект `state` из объемлющей области видимости по существу присоединен к возвращаемому объекту функции `nested`; каждый вызов создает новый, несовпадающий объект `state`, так что обновление состояния одной функции не оказывает влияния на состояние остальных функций. Ниже продолжается предыдущее взаимодействие:

```
>>> G = tester(42)      # Создание нового объекта функции tester,
# который начинает с 42
>>> G('spam')
spam 42
>>> G('eggs')          # Информация о состоянии обновилась на 43
eggs 43
>>> F('bacon')          # Но функция F осталась там, где и была: 3
bacon 3                 # Каждый вызов имеет разную информацию о состоянии
```

В этом смысле нелокальные переменные Python более утилитарны, чем локальные переменные функции, типичные для ряда других языков. В функции замыкания нелокальные переменные являются *данными с множеством копий для каждого вызова*.

Границные случаи

Несмотря на полезность, с нелокальными переменными связан ряд тонкостей, о которых следует знать. Во-первых, в отличие от оператора `global` к моменту выполнения оператора `nonlocal` перечисленным в нем именам действительно должны быть присвоены значения в области видимости объемлющего `def`, иначе возникнет ошибка — их нельзя создавать динамически, присваивая заново во вложенной области видимости. На самом деле они проверяются на этапе определения функций перед вызовом либо объемлющей, либо вложенной функции:

```
>>> def tester(start):
    def nested(label):
        nonlocal state      # Нелокальные переменные уже должны
                            # существовать в объемлющем def!
        state = 0
        print(label, state)
    return nested

SyntaxError: no binding for nonlocal 'state' found
Синтаксическая ошибка: не найдена привязка для нелокальной переменной state
>>> def tester(start):
    def nested(label):
        global state      # Глобальные переменные не обязаны существовать
                            # до их объявления
        state = 0          # Создает имя в области видимости модуля
        print(label, state)
    return nested
>>> F = tester(0)
>>> F('abc')
```

```
abc 0
>>> state
0
```

Во-вторых, оператор `nonlocal` ограничивает поиск в областях видимости только объемлющими `def`; нелокальные переменные не ищутся в глобальной области видимости включающего модуля или во встроенной области видимости за пределами всех `def`, даже если они там находятся:

```
>>> spam = 99
>>> def tester():
    def nested():
        nonlocal spam      # Должна быть в def, а не в модуле!
        print('Current=', spam)
        spam += 1
    return nested
```

SyntaxError: no binding for nonlocal 'spam' found

Синтаксическая ошибка: не найдена привязка для нелокальной переменной `spam`

Такие ограничения обретут смысл, как только вы осознаете, что в противном случае Python вообще не знал бы, в какой объемлющей области видимости создавать совершенно новое имя. Где в предыдущем примере должна присваиваться переменная `spam` — внутри `tester` или во включающем модуле? Поскольку это неоднозначно, Python вынужден распознавать нелокальные переменные на этапе *создания* функций, а не во время *вызыва* функций.

Для чего используются оператор `nonlocal`? Варианты сохранения состояния

Учитывая дополнительную сложность вложенных функций, вас может интересовать, для чего они нужны. Хотя это трудно разглядеть в наших небольших примерах, во многих программах информация о состоянии становится критически важной. Несмотря на то что функции способны возвращать результаты, их локальные переменные обычно не сохраняют остальные значения, которые должны существовать между вызовами. Более того, многочисленные приложения требуют, чтобы такие значения отличались в зависимости от контекста применения.

Ранее уже упоминалось, что в Python доступны разнообразные способы “запоминания” информации между вызовами функций и методов. Невзирая на наличие компромиссов, связанных со всеми способами, оператор `nonlocal` улучшает положение дел для ссылок на переменные из объемлющих областей видимости — `nonlocal` позволяет хранить в памяти множество копий *изменяемого* состояния. Он решает простую задачу сохранения состояния, когда использование классов может быть не оправданным, а глобальные переменные неприменимы, хотя атрибуты функций часто могут исполнять похожие роли более переносимым образом. Ниже приведен обзор возможных вариантов.

Состояние с помощью оператора `nonlocal`: только Python 3.X

Как было показано в предыдущем разделе, представленный далее код позволяет сохранять и модифицировать состояние из объемлющей области видимости. Каждый вызов `tester` создает самодостаточный *пакет изменяемой информации*, а имена пакетов не конфликтуют с любой другой частью программы:

```

>>> def tester(start):
    state = start      # Каждый вызов получает собственное значение state
    def nested(label):
        nonlocal state # Запоминает state из объемлющей области видимости
        print(label, state)
        state += 1      # Нелокальную переменную разрешено изменять
    return nested

>>> F = tester(0)
>>> F('spam')          # Переменная state видима только внутри замыкания
spam 0
>>> F.state
AttributeError: 'function' object has no attribute 'state'
Ошибка атрибута: объект function не имеет атрибута state

```

Нам необходимо объявлять переменные нелокальными, только если они должны изменяться (остальные ссылки на имена из объемлющих областей видимости автоматически сохраняются как обычно), и нелокальные имена по-прежнему не будут видимы за пределами объемлющей функции.

К сожалению, такой код работает только в Python 3.X. Если вы используете Python 2.X, тогда в зависимости от преследуемых целей доступны другие варианты. В следующих трех разделах описаны альтернативы. В коде местами применяются инструменты, которые пока еще не были раскрыты и отчасти предназначены для предварительного обзора, но мы сохраняем примеры простыми, чтобы вы могли попутно сравнивать их друг с другом.

Состояние с помощью глобальных переменных: только одиночная копия

Распространенная рекомендация для достижения в Python 2.X и предшествующих версиях эффекта, подобного nonlocal, предусматривает просто вынесение состояния в глобальную область видимости (включающий модуль):

```

>>> def tester(start):
    global state      # Вынести в модуль, чтобы можно было изменять
    state = start     # global делает возможными изменения
                      # в области видимости модуля
    def nested(label):
        global state
        print(label, state)
        state += 1
    return nested

>>> F = tester(0)
>>> F('spam')  # Каждый вызов инкрементирует разделяемое глобальное состояние
spam 0
>>> F('eggs')
eggs 1

```

Прием в данном случае работает, но требует объявлений global в обеих функциях и предрасположен к конфликтам имен в глобальной области видимости (что, если имя state уже задействовано?). Худшая и более тонкая проблема заключается в том, что он допускает единственную разделяемую копию информации о состоянии в области видимости модуля — если мы вызовем tester снова, то инициируем переустановку переменной state модуля, так что предыдущие вызовы увидят свои переменные state перезаписанными:

```

>>> G = tester(42)      # Сбрасывает единственную копию state в глобальной
                           # области видимости
>>> G('toast')
toast 42
>>> G('bacon')
bacon 43
>>> F('ham')           # Но мой счетчик был перезаписан!
ham 44

```

Как было показано ранее, когда вместо `global` используется оператор `nonlocal` и замыкания вложенных функций, каждый вызов `tester` запоминает собственную уникальную копию объекта `state`.

Состояние с помощью классов: явные атрибуты (предварительный обзор)

Еще одна рекомендация для поддержки изменяемой информации о состоянии в Python 2.X и предшествующих версиях заключается в применении классов с атрибутами, чтобы сделать доступ к информации о состоянии более явным, чем скрытая магия правил поиска в областях видимости. В качестве добавочного преимущества каждый экземпляр класса получает новую копию информации о состоянии как естественный побочный продукт объектной модели Python. Классы также поддерживают наследование, множество линий поведения и другие инструменты.

Мы пока не рассматривали классы подробно, но для сравнения и краткого обзора ранее показанные функции `tester/nested` переделаны в класс, который явно регистрирует состояние в объектах при их создании. Чтобы понять код, необходимо знать, что оператор `def` внутри `class` работает в точности как нормальный `def` за исключением того, что аргумент `self` функции автоматически получает подразумеваемый объект вызова (экземпляр, созданный обращением к самому классу). При обращении к классу автоматически запускается функция по имени `__init__`:

```

>>> class tester:          # Альтернатива на основе класса (см. часть VI)
    def __init__(self, start): # При создании объектов состояние
        self.state = start    # явно сохраняется в новом объекте
    def nested(self, label):
        print(label, self.state) # Явная ссылка на состояние
        self.state += 1         # Изменения также разрешены
>>> F = tester(0)          # Создание экземпляра, вызов __init__
>>> F.nested('spam')       # F передается аргументу self
spam 0
>>> F.nested('ham')
ham 1

```

В классе мы сохраняем *каждый* атрибут явно, изменяется он или на него производится ссылка, и атрибуты доступны за пределами класса. Подобно вложенным функциям и оператору `nonlocal` альтернатива в форме класса поддерживает множество копий сохраненных данных:

```

>>> G = tester(42)          # Каждый экземпляр получает новую копию состояния
>>> G.nested('toast')      # Изменение одного не влияет на остальные
toast 42
>>> G.nested('bacon')
bacon 43
>>> F.nested('eggs')       # Состояние F осталось прежним

```

```

eggs 2
>>> F.state          # Доступ к состоянию возможен извне класса
3

Добавив лишь чуть больше магии (в которую мы углубимся позже в книге), мы могли бы сделать объекты классов выглядящими как вызываемые функции, используя перегрузку операций. Функция __call__ перехватывает прямые обращения к экземпляру, так что вызывать какой-то именованный метод не требуется:

>>> class tester:
    def __init__(self, start):
        self.state = start
    def __call__(self, label): # Перехватывает прямые обращения к экземпляру
        print(label, self.state) # Таким образом, .nested() не требуется
        self.state += 1

>>> H = tester(99)
>>> H('juice')           # Вызывает __call__
juice 99
>>> H('pancakes')
pancakes 100

```

Пока не стоит беспокоиться по поводу деталей показанного кода; он приведен в основном как предварительный обзор с намерением только сравнить классы с замыканиями. Мы будем исследовать классы в части VI, а специфические инструменты для перегрузки операций наподобие `__call__` рассмотрим в главе 30. Здесь важно отметить, что классы способны сделать информацию о состоянии более очевидной за счет того, что они задействуют явное присваивание атрибутов вместо неявного поиска в областях видимости. Вдобавок атрибуты классов всегда допускают изменение и не требуют оператора `nonlocal`, а классы рассчитаны на реализацию объектов с более широкими возможностями, имеющими многочисленные атрибуты и линии поведения.

Наряду с тем, что применение классов для хранения информации о состоянии в целом является хорошим эмпирическим правилом, которому имеет смысл следовать, они также могут оказываться *крайностью* в ситуациях вроде этой, где состояние представляет собой одиночный счетчик. Такие тривиальные варианты состояния встречаются чаще, чем вы могли подумать; в подобных случаях вложенные `def` иногда будут легковеснее классов, особенно когда вы еще не освоили объектно-ориентированное программирование. Более того, существуют сценарии, в которых вложенные `def` на самом деле могут работать *лучше* классов – пример, выходящий далеко за рамки настоящей главы, будет приведен в главе 39, где описаны *декораторы методов*.

Состояние с помощью атрибутов функций: Python 3.X и 2.X

В качестве переносимого и часто более простого варианта хранения состояния мы можем временами достичь эффекта нелокальных переменных с помощью *атрибутов функций* – определяемых пользователем имен, напрямую присоединяемых к функциям. Когда определяемые пользователем атрибуты присоединяются к вложенным функциям, генерируемым объемлющими фабричными функциями, они способны также служить состоянием для каждого вызова с множеством копий и возможностью записи, что очень похоже на нелокальные замыкания и атрибуты классов. Имена определяемых пользователем атрибутов не будут конфликтовать с именами, создаваемыми Python самостоятельно, и как при использовании `nonlocal`, они применяются только

для переменных состояния, которые должны изменяться; ссылки на остальные переменные сохраняются и работают нормально.

Важно отметить, что такая схема *переносима* – подобно классам, но в отличие от nonlocal, атрибуты функций работают в Python 3.X и в Python 2.X. Фактически они доступны, начиная с версии Python 2.1, т.е. гораздо дольше, чем оператор nonlocal из Python 3.X. Поскольку фабричные функции в любом случае создают новую функцию при каждом вызове, дополнительные объекты не требуются – атрибуты новой функции становятся состоянием для вызова во многом таким же образом, как нелокальные переменные, и аналогично ассоциируются со сгенерированной функцией в памяти.

Более того, как и атрибуты классов, атрибуты функций делают возможным доступ к переменным состояния *извне* вложенной функции; посредством nonlocal переменные состояния можно сделать видимыми только внутри вложенного def. Если необходим внешний доступ к счетчику вызовов, то в данной модели это будет простое извлечение атрибута функции.

Ниже показана финальная версия, основанная на методике с атрибутами функций – в ней nonlocal заменяется атрибутом, присоединенным к вложенной функции. На первый взгляд схема может показаться не особенно понятной; доступ к состоянию осуществляется через имя функции, а не простые переменные, и оно должно быть инициализировано после вложенного оператора def. Но результирующий код намного более переносим, разрешает внешний доступ к состоянию и экономит строку кода, т.к. не требует объявления nonlocal:

```
>>> def tester(start):
    def nested(label):
        print(label, nested.state) # Функция nested находится
                                    # в объемлющей области видимости
        nested.state += 1 # Изменяет атрибут, а не саму функцию nested
    nested.state = start # Инициализация состояния после определения функции
    return nested

>>> F = tester(0)
>>> F('spam')           # F - это nested с присоединенным состоянием
spam 0
>>> F('ham')
ham 1
>>> F.state             # Возможен также доступ к состоянию извне функции
2
```

Из-за того, что каждое обращение к внешней функции производит новый объект вложенной функции, эта схема поддерживает изменяемые данные с множеством копий для каждого вызова в точности как нелокальные замыкания и классы – режим использования, который глобальные переменные обеспечить не могут:

```
>>> G = tester(42) # G имеет собственное состояние, состояние F не перезаписывается
>>> G('eggs')
eggs 42
>>> F('ham')
ham 2
>>> F.state          # Состояние доступно и поддерживается для каждого вызова
3
>>> G.state
43
>>> F is G           # Разные объекты функций
False
```

Код полагается на тот факт, что имя функции `nested` представляет собой локальную переменную в области видимости `tester`, включающей в себя `nested`; по существу на нее можно свободно ссылаться внутри `nested`. Код также основывается на том, что изменение объекта на месте не является присваиванием имени; когда инкрементируется `nested.state`, то изменяется часть объекта, на который ссылается `nested`, а не само имя `nested`. Поскольку на самом деле мы не выполняем присваивание имени из объемлющей области видимости, никакие объявления `nonlocal` не требуются.

Атрибуты функций поддерживаются в Python 3.X и 2.X; мы более подробно рассмотрим их в главе 19. Там мы увидим одну важную деталь: в Python 2.X и 3.X применяются соглашения об именовании, гарантирующие отсутствие конфликтов между произвольными именами, которые вы назначаете атрибутам функций, и именами, которые имеют отношение к внутренней реализации, что делает пространство имен эквивалентным области видимости. Помимо субъективных факторов роль атрибутов частично перекрывается с ролью более нового оператора `nonlocal` в Python 3.X, приводя к его избыточности и меньшей переносимости.

Состояние с помощью изменяемых объектов: неотчетливый призрак прошлого Python?

В качестве связанного замечания: в Python 2.X and 3.X возможно также изменять *изменяемый* объект из объемлющей области видимости, не объявляя его имя в `nonlocal`. Скажем, следующий код работает точно так же, как предыдущая версия, является в такой же степени переносимым и обеспечивает изменяемое состояние для каждого вызова:

```
def tester(start):
    def nested(label):
        print(label, state[0])
        # Использует в своих интересах изменение
        state[0] += 1
        # на месте изменяемого объекта
        state = [start]
        # Добавочный синтаксис, глубинная магия?
        return nested
```

Код задействует изменяемость списков и подобно атрибутам функций опирается на тот факт, что изменение объекта на месте не классифицирует имя как локальное. Однако такой код вероятно еще менее понятен, чем атрибуты функций или оператор `nonlocal` из Python 3.X – он демонстрирует методику, которая предшествовала даже атрибутам функций, и в наши дни, похоже, находится где-то между искусственной поделкой и черной магией! Возможно, лучше использовать именованные атрибуты функций, чем списки и числовые смещения подобным образом, хотя такие решения могут встречаться в коде, с которым вам придется работать.

Подытожим: глобальные переменные, нелокальные переменные, классы и атрибуты функций предлагают варианты сохранения изменяемого состояния. Глобальные переменные поддерживают лишь одиночную копию разделяемых данных; нелокальные переменные могут изменяться только в Python 3.X; классы требуют базовых знаний объектно-ориентированного программирования; а классы и атрибуты функций предоставляют переносимые решения, которые делают возможным доступ к состоянию напрямую извне самого запоминающего состояния вызываемого объекта. Как обычно, наилучший инструмент для программы определяется целями, преследуемыми программой.

В главе 39 мы снова вернемся к представленным здесь вариантам поддержки состояния при исследовании более реалистичного контекста – декораторов, которые яв-

ляются инструментом, по своей природе связанным с многоуровневым сохранением состояния. Варианты поддержки состояния обладают дополнительными характеристиками, влияющими на их выбор (например, производительность), которые в настоящей главе не раскрыты из-за нехватки места (в главе 21 мы покажем, как измерять скорость выполнения кода). А теперь пришло время заняться режимами передачи аргументов.

Что потребует внимания: настройка open

Рассмотрим еще один пример замыканий в работе. Предположим, что необходимо заменить встроенный вызов open специальной версией, как упоминалось во врезке “Разрушение мироздания в Python 2.X” ранее в главе. Если специальная версия должна вызывать оригинал, тогда она обязана сохранить его перед изменением и удерживать для дальнейшего применения – классический сценарий с сохранением состояния. Кроме того, если желательно поддерживать множество настроенных версий для той же самой функции, то глобальные переменные не помогут: нас интересует состояние для каждой настроенной версии.

В следующем далее коде Python 3.X, находящемся в файле makeopen.py, иллюстрируется один из способов решения задачи (в Python 2.X понадобится изменить имя встроенной области видимости и вывод). В нем с использованием замыкания вложенной области видимости запоминается значение для применения в будущем, не полагаясь на глобальные переменные, которые могут привести к конфликту имен и допускают только одно значение, а также не прибегая к услугам класса, который может потребовать написания большего объема кода, чем здесь оправдано:

```
import builtins

def makeopen(id):
    original = builtins.open
    def custom(*pargs, **kargs):
        print('Custom open call %r:' % id, pargs, kargs)
        return original(*pargs, **kargs)
    builtins.open = custom
```

Чтобы изменить вызов open для каждого модуля внутри процесса, в коде ему присваивается специальная версия, написанная с помощью вложенного оператора def, после сохранения оригинала в объемлющей области видимости, так что настроенная версия сможет обращаться к нему позже. Код еще служит и предварительной демонстрацией, т.к. при сборке и дальнейшей распаковке произвольных позиционных и ключевых аргументов, предназначенных для open, он полагается на формы аргументов, снабженных звездочкой – тема следующей главы. Однако большая часть магии скрыта в замыканиях вложенных областей видимости: специальная версия open, найденная правилами поиска в областях видимости, удерживает в себе оригинал для будущего использования:

```
>>> F = open('script2.py')      # Вызов встроенной функции open в builtins
>>> F.read()
'import sys\nprint(sys.path)\n*x = 2\nprint(x ** 32)\n'
>>> from makeopen import makeopen # Импортирование функции переустановки open
>>> makeopen('spam')           # Специальная версия open вызывает встроенную
                               # версию open
>>> F = open('script2.py')      # Вызов специальной версии open в builtins
Custom open call 'spam': ('script2.py',)
>>> F.read()
'import sys\nprint(sys.path)\n*x = 2\nprint(x ** 32)\n'
```

Поскольку каждая настроенная версия запоминает первую встроенную версию в собственной объемлющей области видимости, по своей природе они могут даже вкладываться способами, которые глобальные переменные не способны поддерживать — каждый вызов функции замыкания `makeopen` запоминает собственные версии `id` и `original`, так что допускается запускать множество настроенных версий:

```
>>> makeopen('eggs')          # Вложенные настроенные версии тоже работают!
>>> F = open('script2.py')    # Из-за того, что каждая сохраняет
                             # собственное состояние
Custom open call 'eggs': ('script2.py',) {}
Custom open call 'spam': ('script2.py',) {}
>>> F.read()
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'


```

В том виде, как есть, наша функция просто добавляет к встроенной функции вывод трассировки возможно вложенного вызова, но общая методика способна иметь и другие применения. Эквивалент на основе классов может требовать большего объема кода, потому что ему необходимо сохранять значения `id` и `original` явно в атрибутах объекта — но для его реализации нужны знания, которые пока отсутствуют, а потому отложим данную тему до части VI и приведем лишь обзор:

```
import builtins

class makeopen:      # См. часть VI: вызов захватывает self()
    def __init__(self, id):
        self.id = id
        self.original = builtins.open
        builtins.open = self
    def __call__(self, *pargs, **kargs):
        print('Custom open call %r:' % self.id, pargs, kargs)
        return self.original(*pargs, **kargs)


```

Здесь следует отметить, что классы могут быть более явными, но также требовать написания большего объема кода, когда единственной целью является сохранение состояния. Мы увидим дополнительные сценарии использования позже, особенно когда займемся исследованием *декораторов* в главе 39, где обнаружится, что замыкания в определенных ролях предпочтительнее классов.

Резюме

В главе вы изучили одну из двух ключевых концепций, связанных с функциями: *области видимости*, которые определяют, каким образом ищутся переменные, когда на них производится ссылка. Вы узнали, что переменные считаются локальными в определениях функций, в которых им выполняется присваивание, если только они специально не объявлены как глобальные или нелокальные. Мы также исследовали ряд более сложных концепций, касающихся областей видимости, включая области видимости вложенных функций и атрибуты функций. Наконец, мы взглянули на несколько универсальных идей проектирования, в том числе избегание применения глобальных имен и межфайловых изменений.

В следующей главе мы продолжим наш тур по функциям рассмотрением второй ключевой концепции, связанной с функциями: передачи аргументов. Как вы увидите, аргументы передаются в функцию по присваиванию, но Python также предлагает инструменты, которые позволяют функциям быть гибкими в плане передачи элементов. Но прежде чем двигаться дальше, закрепите пройденный материал этой главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Что выведет следующий код и почему?

```
>>> X = 'Spam'  
>>> def func():  
    print(X)  
  
>>> func()
```

2. Что выведет следующий код и почему?

```
>>> X = 'Spam'  
>>> def func():  
    X = 'NI'  
  
>>> func()  
>>> print(X)
```

3. Что выведет следующий код и почему?

```
>>> X = 'Spam'  
>>> def func():  
    X = 'NI'  
    print(X)  
  
>>> func()  
>>> print(X)
```

4. Что выведет следующий код и почему?

```
>>> X = 'Spam'  
>>> def func():  
    global X  
    X = 'NI'  
  
>>> func()  
>>> print(X)
```

5. Что выведет следующий код и почему?

```
>>> X = 'Spam'  
>>> def func():  
    X = 'NI'  
    def nested():  
        print(X)  
    nested()  
  
>>> func()  
>>> X
```

6. Что выведет следующий код в Python 3.X и почему?

```
>>> def func():  
    X = 'NI'  
    def nested():  
        nonlocal X  
        X = 'Spam'  
    nested()  
    print(X)  
  
>>> func()
```

7. Назовите три или большее количество способов сохранения информации о состоянии в функции Python.

Проверьте свои знания: ответы

1. Выводится 'Spam', т.к. функция ссылается на глобальную переменную из включающего модуля (поскольку присваивание переменной в функции не производится, переменная считается глобальной).
2. Выводится снова 'Spam', потому что присваивание переменной внутри функции делает ее локальной и фактически скрывает глобальную переменную с таким же именем. Оператор `print` обнаруживает переменную неизмененной в глобальной области видимости (модуля).
3. Выводится 'NI' в одной строке и 'Spam' в другой, т.к. ссылка на переменную внутри функции находит присвоенную локальную переменную, а ссылка в операторе `print` – глобальную переменную.
4. На этот раз выводится 'NI', поскольку объявление `global` заставляет переменную, присваиваемую внутри функции, ссылаться на переменную в объемлющей глобальной области видимости.
5. В данном случае выводится снова 'NI' в одной строке и 'Spam' в другой, потому что оператор `print` во вложенной функции находит имя в локальной области видимости объемлющей функции, а отображение в конце находит переменную в глобальной области видимости.
6. Выводится 'Spam', поскольку оператор `nonlocal` (доступный в Python 3.X, но не в Python 2.X) означает, что присваивание `X` внутри вложенной функции изменяет `X` в локальной области видимости объемлющей функции. Без этого оператора присваивание классифицировало бы `X` как локальную переменную во вложенной функции, делая ее другой переменной; тогда код выводил бы 'NI'.
7. Хотя значения локальных переменных исчезают, когда происходит возврат из функции, вы можете заставить функцию Python сохранять информацию о состоянии за счет использования разделяемых глобальных переменных, ссылок внутри вложенных функций на переменные из области видимости объемлющих функций или стандартных значений аргументов. Атрибуты функций могут иногда позволить состоянию присоединяться к самим функциям вместо выполнения поиска в областях видимости. Еще одна альтернатива, применение классов и объектно-ориентированного программирования, временами поддерживает сохранение состояния лучше, чем любые методики, основанные на областях видимости, потому что делает его явным посредством присваивания атрибутов; мы исследуем этот вариант в части VI.

Аргументы

В главе 17 обсуждались детали *областей видимости* Python – мест, где определяются и ищутся переменные. Вы узнали, что место определения имени в коде во многом обуславливает его смысл. В этой главе тема функций продолжается исследованием концепций *передачи аргументов* – способа отправки функциям объектов как входных данных. Вы увидите, что аргументы (называемые также параметрами) присваиваются именам в функции, но они имеют большее отношение к ссылкам на объекты, чем к областям видимости переменных. Вы обнаружите, что Python предоставляет добавочные инструменты, такие как ключевые аргументы, аргументы со стандартными значениями, а также собиратели и экстракторы произвольных аргументов, которые делают возможной большую гибкость в способе передачи аргументов функции.

Основы передачи аргументов

Ранее в текущей части книги упоминалось, что аргументы передаются по *присваиванию*. В итоге возникает несколько последствий, не всегда очевидных новичкам, которые будут раскрыты в настоящем разделе. Ниже приведено краткое изложение ключевых аспектов передачи аргументов функциям.

- Аргументы передаются путем автоматического присваивания объектов именам локальных переменных. Аргументы функций, т.е. ссылки на (возможно) разделяемые объекты, отправленные вызывающим кодом, представляют собой еще один пример присваивания Python в действии. Поскольку ссылки реализованы в виде указателей, все аргументы в действительности передаются через указатели. Объекты, передаваемые как аргументы, никогда автоматически не копируются.
- Присваивание именам аргументов внутри функции не затрагивает вызывающий код. Когда функция выполняется, имена аргументов в заголовке функции становятся новыми локальными именами в области видимости функции. Никакого совмещения имен аргументов функции и имен переменных в области видимости вызывающего кода не происходит.
- Модификация внутри функции аргумента, являющегося изменяемым объектом, может затронуть вызывающий код. С другой стороны, так как аргументам просто присваиваются передаваемые объекты, в функциях можно модифицировать переданные изменяемые объекты на месте, в результате оказывая влияние на вызывающий код. Изменяемые аргументы могут служить входными и выходными данными для функций.

За дополнительными сведениями о *ссылках* обращайтесь в главу 6; все, что там приводилось, также применимо к аргументам функций, хотя присваивание именам аргументов происходит автоматически и неявно.

Схема передачи по присваиванию Python – не совсем то же самое, что и ссылочные параметры C++, но на практике она очень похожа на модель передачи аргументов в C (и других языках).

- Неизменяемые аргументы фактически передаются “по значению”. Объекты, подобные целым числам и строкам, передаются по ссылке на объекты, а не путем копирования, но из-за того, что модифицировать на месте неизменяемые объекты невозможно, эффект во многом похож на создание копий.
- Изменяемые аргументы фактически передаются “по указателю”. Объекты вроде списков и словарей также передаются по ссылке на объекты, что аналогично способу передачи массивов как указателей в языке С – изменяемые объекты можно модифицировать на месте в функции почти как массивы в C.

Разумеется, если вы никогда не использовали язык C, то режим передачи аргументов в Python вам покажется еще более простым – он влечет за собой только присваивание объектов именам и работает одинаково для изменяемых и неизменяемых объектов.

Аргументы и разделяемые ссылки

Чтобы оценить характеристики передачи аргументов в работе, рассмотрим следующий код:

```
>>> def f(a):      # a присваивается (устанавливается в ссылку на) переданный объект
    a = 99        # Изменяет только локальную переменную a

>>> b = 88
>>> f(b)         # a и b первоначально ссылаются на 88
>>> print(b)     # Значение b не изменилось
88
```

В приведенном примере переменной a присваивается объект 88 в момент, когда функция вызывается посредством `f(b)`, но a существует только внутри вызванной функции. Изменение a внутри функции никак не воздействует на место, где функция вызывалась; оно просто переустанавливает локальную перемененную a в совершенно другой объект.

Вот что означает отсутствие *совмещения* имен – присваивание имени аргумента внутри функции (например, `a=99`) не изменяет волшебным образом переменную вроде b в области видимости, где вызывалась функция. Имена аргументов могут изначально разделять передаваемые объекты (по существу они являются указателями на эти объекты), но только временно, когда функция вызывается впервые. После повторного присваивания имени аргумента связь разрывается.

По крайней мере, так происходит в случае присваивания самим именам аргументов. Когда аргументам передаются *изменяемые* объекты, подобные спискам и словарям, мы должны осознавать, что изменения на месте таких *объектов* могут продолжить свое существование после завершения функции, а потому оказывать воздействие на вызывающий код. Ниже представлен пример, демонстрирующий описанное поведение:

```
>>> def changer(a, b):      # Аргументам присваиваются ссылки на объекты
    a = 2                    # Изменяет только значение локального имени
    b[0] = 'spam'            # Изменяет разделяемый объект на месте

>>> X = 1
>>> L = [1, 2]              # Вызывающий код:
>>> changer(X, L)          # Передача неизменяемого и изменяемого объектов
>>> X, L                  # X остается прежним, L отличается!
(1, ['spam', 2])
```

Функция `changer` присваивает значения самому аргументу `a` и компоненту `объекта`, на который ссылается аргумент `b`. Эти два присваивания внутри функции лишь слегка отличаются по синтаксису, но приводят к совершенно разным результатам.

- Поскольку `a` – имя локальной переменной в области видимости функции, первое присваивание не влияет на вызывающий код; оно просто модифицирует локальную переменную `a`, чтобы `a` ссылалась на другой объект, и не изменяет привязку имени `X` в области видимости вызывающего кода. Ситуация такая же, как в предыдущем примере.
- Аргумент `b` тоже является именем локальной переменной, но ему передан изменяемый объект (список, на который ссылается `L` в области видимости вызывающего кода). Так как второе присваивание представляет собой изменение объекта на месте, результат присваивания `b[0]` в функции оказывает воздействие на значение `L` после возврата управления из функции.

На самом деле второй оператор присваивания в `changer` не изменяет `b` – он модифицирует часть объекта, на который ссылается `b` в текущий момент. Такое изменение на месте влияет только на вызывающий код, потому что модифицированный объект остается существовать после вызова функции. Имя `L` тоже не изменяется (оно по-прежнему ссылается на тот же самый модифицированный объект), но выглядит так, будто бы имя `L` после вызова стало другим, потому что значение, на которое оно ссылается, было изменено внутри функции. Фактически список по имени `L` служит входными и выходными данными для функции.

На рис. 18.1 иллюстрируются привязки имя/объект, которые существуют непосредственно после того, как функция была вызвана, и перед выполнением ее кода.

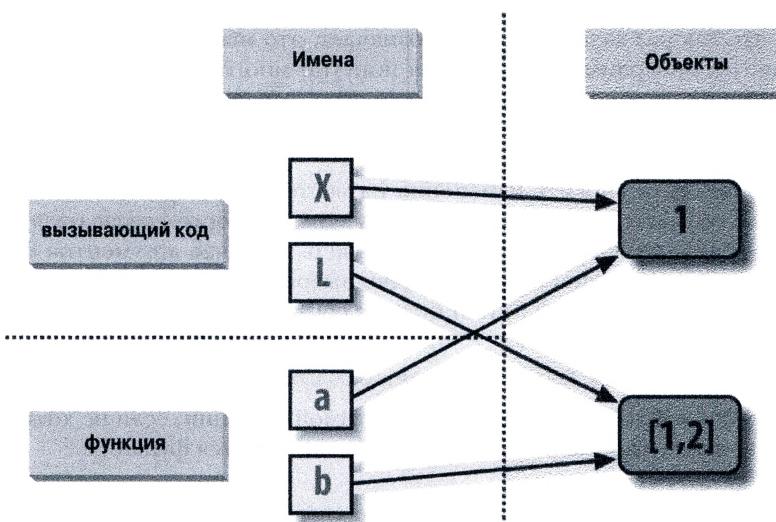


Рис. 18.1. Ссылки: аргументы. Поскольку аргументы передаются по присваиванию, имена аргументов в функции могут разделять объекты с переменными в области видимости вызова. Следовательно, изменения на месте аргументов с изменяемыми объектами в функции могут влиять на вызывающий код. Когда функция вызывается, а и b внутри функции первоначально ссылаются на объекты, на которые ссылаются переменные X и L. Модификация списка через переменную b делает переменную L выглядящей по-другому после возврата управления из вызова

Если рассмотренный пример все еще сбивает с толку, тогда может помочь тот факт, что результат автоматических присваиваний передаваемых аргументов будет таким же, как и выполнение серии простых операторов присваивания. Что касается первого аргумента, то присваивание никак не воздействует на вызываемый код:

```
>>> X = 1
>>> a = X      # Разделяют тот же самый объект
>>> a = 2      # Переустанавливает только a, значением X по-прежнему будет 1
>>> print(X)
1
```

Однако присваивание через второй аргумент оказывает влияние на переменную в вызывающем коде, т.к. оно является изменением объекта на месте:

```
>>> L = [1, 2]
>>> b = L          # Разделяют тот же самый объект
>>> b[0] = 'spam'  # Изменение на месте: L тоже увидит изменение
>>> print(L)
['spam', 2]
```

Вспомнив обсуждение разделяемых изменяемых объектов из глав 6 и 9, вы опознаете данное явление в действии: модификация изменяемого объекта на месте может оказывать влияние на другие ссылки на этот объект. Здесь эффект заключается в том, что аргументы работают подобно входным и выходным данным функции.

Избегайте модификации изменяемых аргументов

Подобное поведение модификации на месте изменяемых аргументов ошибкой не является — просто именно так работает передача аргументов в Python, которая еще и оказывается очень удобной на практике. Аргументы обычно передаются функциям по ссылке, что как раз желательно. Это означает, что мы можем передавать крупные объекты внутри программы, не создавая попутно многочисленные копии, и в ходе дела легко обновлять такие объекты. Фактически, как будет показано в части VI, при обновлении состояния объекта модель классов Python основывается на изменении переданного аргумента `self` на месте.

Тем не менее, если вносить изменения на месте внутри функций, воздействуя на передаваемые объекты, нежелательно, тогда можно просто создавать явные копии изменяемых объектов, как объяснялось в главе 6. В случае аргументов функций мы всегда можем создать копию списка в точке вызова с помощью инструмента вроде `list.copy` (начиная с Python 3.3) или пустого среза:

```
L = [1, 2]
changer(X, L[:])      # Передача копии, так что L не изменяется
```

Мы можем также создавать копию внутри самой функции, если не хотим изменять передаваемые объекты независимо от того, как вызывается функция:

```
def changer(a, b):
    b = b[:]           # Создание копии входного списка,
                      # чтобы не влиять на вызывающий код
    a = 2
    b[0] = 'spam'     # Изменяет только копию списка
```

Обе схемы копирования не мешают функции изменять объект — они всего лишь не допускают влияния изменений на вызывающий код. Чтобы действительно предотвратить изменения, мы всегда можем преобразовать в неизменяемые объекты, модифи-

кация которых вызывает проблему. Скажем, попытка изменения кортежей приводит к генерации исключения:

```
L = [1, 2]
changer(X, tuple(L)) # Передача кортежа, так что изменения приведут к ошибкам
```

В коде применяется встроенная функция `tuple`, которая строит новый кортеж из всех элементов в последовательности (на самом деле в любом итерируемом объекте). Решение выглядит чрезмерным. Поскольку функцию приходится писать так, чтобы она никогда не изменяла передаваемые аргументы, наложенных на нее ограничений может оказаться больше, чем должно быть, и потому в целом такого подхода следует избегать (изменение аргументов вполне может пригодиться в будущем). Использование указанной методики также приведет к тому, что функция будет лишена возможности вызывать на аргументе любые методы, специфичные для списков, включая те, которые не изменяют объект на месте.

Здесь важно помнить, что функции могут обновлять передаваемые им изменяемые объекты, подобные спискам и словарям. Когда такой эффект ожидается, он не обязательно считается проблемой и часто преследует полезные цели. Более того, функции, модифицирующие передаваемые им изменяемые объекты на месте, возможно, были спроектированы и предназначены делать именно это — вполне вероятно, что изменение является частью четко определенного API-интерфейса, который не должен нарушаться созданием копий.

Однако вы обязаны знать о таком свойстве — если объекты изменяются неожиданно для вас, тогда проверьте, несет ли за это ответственность вызываемая функция, и при необходимости создавайте копии при передаче объектов.

Эмуляция выходных параметров и множественных результатов

Мы уже обсуждали оператор `return` и применяли его в нескольких примерах. Существует еще один способ использования этого оператора: из-за того, что `return` способен отправлять обратно объект любого вида, он может возвращать *множественные значения*, упаковывая их в кортеж или коллекцию другого типа. На самом деле, хотя Python не поддерживает то, что в некоторых языках называется передачей аргументов по ссылке, мы обычно можем эмулировать его, возвращая кортежи и присваивая результаты исходным именам аргументов в вызывающем коде:

```
>>> def multiple(x, y):
    x = 2                      # Изменяет только локальные имена
    y = [3, 4]
    return x, y                 # Возвращение множества новых значений в кортеже

>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)    # Присваивание результатов именам в вызывающем коде
>>> X, L
(2, [3, 4])
```

Выглядит так, будто код возвращает два значения, но в действительности значение только одно — двухэлементный кортеж, в котором опущены необязательные окружающие круглые скобки. После возврата управления из вызова можно применить присваивание кортежей для распаковки частей возвращенного кортежа. (Если вы забыли, почему оно работает, перечитайте разделы “Кортежи” в главах 4 и 9, а также раздел “Операторы присваивания” в главе 11.) Совокупный эффект такого кодового шаблона

заключается в том, что он отправляет обратно множественные результаты и эмулирует *выходные параметры* в других языках посредством явных присваиваний. Здесь *X* и *L* изменяются после вызова, но лишь потому, что так реализовано в коде.



Распаковка аргументов в Python 2.X. В предыдущем примере производилась распаковка кортежа, возвращенного функцией, с помощью присваивания кортежей. В Python 2.X можно также автоматически распаковывать кортежи в аргументах, передаваемых в функцию. В Python 2.X (только) функция, определенная со следующим заголовком:

```
def f((a, (b, c))):
```

может быть вызвана с кортежами, которые имеют ожидаемую структуру: `f((1, (2, 3)))` присваивает *a*, *b* и *c* значения 1, 2 и 3 соответственно. Естественно, передаваемый кортеж также может быть объектом, созданным перед вызовом (`f(T)`). Представленный синтаксис `def` больше не поддерживается в Python 3.X. Взамен функция должна быть написана так:

```
def f(T): (a, (b, c)) = T
```

для распаковки в явном операторе присваивания. Такая явная форма работает и в Python 3.X, и в Python 2.X. Как сообщают, распаковка аргументов является малоизвестным и редко используемым средством в Python 2.X (помимо кода, в котором она применяется!). Кроме того, заголовок функции в Python 2.X поддерживает только *кортежную* форму присваивания последовательности; более универсальные присваивания последовательностей (например, `def f((a, [b, c])):`) вызывают в Python 2.X синтаксические ошибки, а также требуют явной формы присваивания, обязательной в Python 3.X. И наоборот, произвольные последовательности в вызове успешно сопоставляются с кортежами в заголовке (скажем, `f((1, [2, 3])), f((1, "ab"))`).

Синтаксис распаковки кортежей в аргументах также запрещен в списках аргументов функции `lambda` в Python 3.X: за примером распаковки для `lambda` обращайтесь во врезку “Что потребует внимания: списковые включения и тар” в главе 20. Несколько противоречиво, но распаковывающее присваивание кортежей по-прежнему автоматически выполняется для целей циклов `for` в Python 3.X; примеры ищите в главе 13.

Специальные режимы сопоставления аргументов

Как было показано, аргументы в Python всегда передаются по *присваиванию*; именам в заголовке `def` присваиваются переданные объекты. Тем не менее, помимо этой модели Python предлагает дополнительные инструменты, которые изменяют способ *сопоставления* объектов-аргументов в вызове с именами аргументов в заголовке до момента присваивания. Все инструменты такого рода необязательны, но они позволяют писать функции, которые поддерживают более гибкие шаблоны вызова, к тому же встречаются библиотеки, которые их требуют.

По умолчанию аргументы сопоставляются по *позиции*, слева направо, и необходимо передавать ровно столько аргументов, сколько есть имен аргументов в заголовке функции. Однако можно также задавать сопоставление по имени, предоставлять стандартные значения и использовать собиратели для добавочных аргументов.

Основы сопоставления аргументов

Прежде чем углубляться в детали синтаксиса, следует подчеркнуть, что специальные режимы необязательны и имеют дело только с сопоставлением объектов с именами; лежащим в основе механизмом после того, как сопоставление произошло, по-прежнему является присваивание. В действительности некоторые инструменты такого рода предназначены в большей степени для тех, кто создает библиотеки, а не для разработчиков приложений. Но поскольку вы можете столкнуться с данными режимами, даже если сами не применяете их в своем коде, ниже представлен краткий обзор доступных инструментов.

Позиционные: сопоставляются слева направо

Нормальный сценарий, который мы главным образом использовали до сих пор, предусматривает сопоставление переданных значений аргументов с именами аргументов в заголовке функции по позиции, слева направо.

Ключевые: сопоставляются по имени аргумента

В качестве альтернативы в вызывающем коде можно указывать, какой аргумент в функции получает значение, за счет применения имени аргумента в вызове посредством синтаксиса `имя=значение`.

Стандартные: указывают значения для необязательных аргументов, которым значения не передавались

Сами функции могут задавать стандартные значения для аргументов, которые они получат, если в вызове передается слишком мало значений, снова с использованием синтаксиса `имя=значение`.

Сбор переменного количества аргументов: собирает произвольно много позиционных и ключевых аргументов

В функциях могут применяться специальные аргументы, предваренные одним или двумя символами `*`, для сбора произвольного количества возможных добавочных аргументов. Такую возможность часто называют *переменным количеством аргументов* (*varargs*) в честь списка аргументов переменной длины в языке С; в Python аргументы собираются в нормальный объект.

Распаковка переменного количества аргументов: передает произвольно много позиционных и ключевых аргументов

В вызывающем коде синтаксис `*` можно также использовать для распаковки коллекций аргументов в отдельные аргументы. Это противоположность `*` в заголовке функции – синтаксис `*` в заголовке означает сбор произвольно большого числа аргументов, тогда как в вызове он означает распаковку произвольно большого количества аргументов и их передачу по отдельности как обособленных значений.

Аргументы с передачей только по ключевым словам: аргументы, которые должны передаваться по имени

В Python 3.X (но не в Python 2.X) функции также допускают указание аргументов, которые должны передаваться по имени с помощью ключевых аргументов, а не по позиции. Такие аргументы обычно применяются для определения конфигурационных параметров в дополнение к фактическим аргументам.

Синтаксис сопоставления аргументов

В табл. 18.1 приведена сводка по синтаксису, который задействует специальные режимы сопоставления аргументов.

Таблица 18.1. Формы сопоставления аргументов функций

Синтаксис	Местоположение	Интерпретация
<code>func(значение)</code>	Вызывающий код	Нормальный аргумент: сопоставляется по позиции
<code>func(имя=значение)</code>	Вызывающий код	Ключевой аргумент: сопоставляется по имени
<code>func(*итерируемый_объект)</code>	Вызывающий код	Передает все объекты в <i>итерируемом_объекте</i> как отдельные позиционные аргументы
<code>func(**словарь)</code>	Вызывающий код	Передает все пары ключ/значение в <i>словаре</i> как отдельные ключевые аргументы
<code>def func(имя)</code>	Функция	Нормальный аргумент: сопоставляется с любым переданным значением по позиции или по имени
<code>def func(имя=значение)</code>	Функция	Стандартное значение аргумента, если значение в вызове не передавалось
<code>def func(*имя)</code>	Функция	Сопоставляет и собирает оставшиеся позиционные аргументы в кортеж
<code>def func(**имя)</code>	Функция	Сопоставляет и собирает оставшиеся ключевые аргументы в словарь
<code>def func(*остальные, имя)</code>	Функция	Аргументы, которые должны передаваться в вызовах только по ключевому слову (Python 3.X)
<code>def func(*, имя=значение)</code>	Функция	Аргументы, которые должны передаваться в вызовах только по ключевому слову (Python 3.X)

Специальные режимы сопоставления подразделяются на вызовы и определения функций следующим образом.

- В вызове функции (первые четыре строки табл. 18.1) простые значения сопоставляются по позиции, но использование формы `имя=значение` сообщает Python о необходимости применения взамен сопоставления по именам аргументов; аргументы такого вида называются **ключевыми**. Использование конструкции `*итерируемый_объект` или `**словарь` в вызове делает возможной упаковку произвольно большого количества позиционных или ключевых аргументов в последовательности (и другие итерируемые объекты) и словари соответственно и распаковку их как индивидуальных аргументов, когда они передаются функции.
- В заголовке функции (остальные строки табл. 18.1) простое имя сопоставляется по позиции или по имени в зависимости от того, каким образом вызывающий

код передает его, но форма `имя=значение` указывает стандартное значение. Форма `*имя` собирает любые добавочные не прошедшие сопоставление позиционные аргументы в кортеж, а форма `**имя` собирает добавочные ключевые аргументы в словарь. В Python 3.X любые нормальные имена аргументов или имена аргументов со стандартными значениями, следующие за конструкцией `*имя` или просто `*`, являются аргументами с передачей только по ключевым словам и должны передаваться в вызовах по ключевому слову.

Вероятно, чаще всего в коде Python применяются ключевые аргументы и стандартные значения. Ранее в книге мы неформально использовали то и другое.

- Мы уже применяли ключевые слова для указания параметров в функции `print` из Python 3.X, но они более универсальны — ключевые слова позволяют обозначать любой аргумент его именем, чтобы делать вызовы информативнее.
- Мы встречали ранее и стандартные значения как способ передачи значений из области видимости объемлющей функции, но данный инструмент также более универсален — он позволяет делать любой аргумент необязательным, снабжая его стандартным значением в определении функции.

Вы увидите, что комбинация стандартных значений в заголовке функции и ключевых слов в последующем вызове дает возможность выбирать, какие стандартные значения должны быть переопределены.

Выражаясь кратко, специальные режимы сопоставления аргументов позволяют проявлять достаточную свободу в отношении того, сколько аргументов должны быть переданы функции. Если в функции указаны стандартные значения, тогда они используются в случае передачи *слишком малого количества* аргументов. Если в функции применяются формы `*` списка с переменным количеством аргументов, то можно спокойно передавать *слишком много* аргументов; имена с `*` собирают добавочные аргументы в структурах данных для обработки внутри функции.

Особенности использования специальных режимов сопоставления

Если вы решили использовать и комбинировать специальные режимы сопоставления аргументов, тогда Python потребует соблюдать описанные далее правила упорядочения для необязательных компонентов.

- В вызове функции аргументы должны указываться в следующем порядке: любые позиционные аргументы (`значение`), за ними комбинация любых ключевых аргументов (`имя=значение`) и формы `*итерируемый_объект`, а затем форма `**словарь`.
- В заголовке функции аргументы должны указываться в следующем порядке: любые нормальные аргументы (`имя`), за ними любые стандартные аргументы (`имя=значение`), далее форма `*имя` (или `*` в Python 3.X), затем аргументы с передачей только по ключевым словам `имя` или `имя=значение` (в Python 3.X) и, наконец, форма `**имя`.

В вызове и заголовке функции форма `**аргументы`, когда присутствует, должна появляться последней. Если вы смешаете аргументы в любом другом порядке, то получите синтаксическую ошибку, потому что комбинации могут быть неоднозначными. Шаги, внутренне выполняемые Python для сопоставления аргументов перед присваиванием, могут быть грубо описаны следующим образом.

1. Присваивание неключевых аргументов по позиции.
2. Присваивание ключевых аргументов по совпадающим именам.
3. Присваивание добавочных неключевых аргументов кортежу *имя.
4. Присваивание добавочных ключевых аргументов словарю **имя.
5. Присваивание стандартных значений неприсвоенным аргументам в заголовке.

Затем Python проверяет, передается ли каждому аргументу только одно значение; если нет, тогда возникает ошибка. Когда сопоставление завершено, Python присваивает именам аргументов переданные для них объекты.

Действительный алгоритм сопоставления, который применяется Python, несколько сложнее (скажем, он обязан также учитывать аргументы с передачей только по ключевым словам в Python 3.X), поэтому за более точным описанием обращайтесь в руководство по языку Python. Читать его не обязательно, но отслеживание алгоритма сопоставления аргументов в Python может содействовать пониманию ряда запутанных ситуаций, особенно когда режимы смешиваются.



В Python 3.X имена аргументов в заголовке функции могут также снабжаться *аннотациями*, указываемыми как `имя: значение` (или `имя: значение=стандартное_значение`, когда присутствуют стандартные значения). Это просто добавочный синтаксис для аргументов, который не дополняет и не изменяет описанные здесь правила упорядочения аргументов. Сама функция может также иметь значение аннотации, заданной как `def f () -> значение`. Python присоединяет значения аннотаций к объекту функции. Дополнительные сведения ищите в обсуждении аннотирования функций в главе 19.

Примеры ключевых слов и стандартных значений

В коде все выглядит проще, чем может вытекать из предшествующих описаний. Если вы не используете какой-то специальный синтаксис сопоставления, то Python сопоставляет имена по позиции слева направо подобно большинству других языков. Например, если вы определили функцию, которая требует трех аргументов, тогда должны вызывать ее с тремя аргументами:

```
>>> def f(a, b, c): print(a, b, c)
>>> f(1, 2, 3)
1 2 3
```

Мы передаем аргументы по позиции – `a` соответствует 1, `b` – 2 и т.д. (сопоставление работает одинаково в Python 3.X и 2.X, но в Python 2.X для кортежа отображаются добавочные круглые скобки, потому что мы снова применяем вызовы `print` из Python 3.X).

Ключевые слова

Тем не менее, в Python вы можете быть более точными в отношении того, что происходит при вызове функции. Ключевые аргументы делают возможным сопоставление по имени, а не по позиции. Ниже используется та же самая функция:

```
>>> f(c=3, b=2, a=1)
1 2 3
```

Здесь `c=3` означает отправку значения 3 аргументу по имени `c`. Более формально Python сопоставляет имя `c` в вызове с аргументом по имени `c` в заголовке определения

функции и затем передает этому аргументу значение 3. Совокупный эффект данного вызова такой же, как у предыдущего, но обратите внимание, что когда применяются ключевые слова, порядок следования аргументов слева направо несущественен, поскольку аргументы сопоставляются по имени, а не по позиции. В одном вызове разрешено даже комбинировать позиционные и ключевые аргументы. В таком случае сначала сопоставляются все позиционные аргументы слева направо в заголовке, а затем ключевые аргументы сопоставляются по имени:

```
>>> f(1, c=3, b=2) # a получает 1 по позиции, значения для b и c передаются по имени  
1 2 3
```

Увидев впервые вызов подобного рода, многие задаются вопросом, для чего мог бы использоваться такой инструмент. Ключевые слова обычно исполняют две роли в Python. Первая роль связана с тем, что они делают вызовы самодокументированными (при условии применения осмысленных имен, а не a, b и c!). Скажем, вызов следующего вида:

```
func(name='Bob', age=40, job='dev')
```

выглядит более значащим по сравнению с вызовом, содержащим три разделенных запятыми значения, особенно в крупных программах — ключевые слова служат метками для данных в вызове. Вторая роль ключевых слов — использование их в сочетании со стандартными значениями, что мы рассмотрим далее.

Стандартные значения

Стандартные значения упоминались ранее при обсуждении областей видимости вложенных функций. Выражаясь кратко, стандартные значения позволяют делать избранные аргументы функции необязательными; если значение для аргумента не было передано, то перед выполнением функции ему присваивается стандартное значение. Например, вот функция с одним обязательным аргументом и двумя аргументами, имеющими стандартные значения:

```
>>> def f(a, b=2, c=3): print(a, b, c)    # Аргумент a обязательный,  
                                         # b и c необязательны
```

При вызове такой функции мы обязаны предоставить значение для a, либо по позиции, либо по ключевому слову; однако передача значений для b и c необязательна. Если мы не передадим значения b и c, тогда они получат стандартные значения 2 и 3 соответственно:

```
>>> f(1)          # Использование стандартных значений  
1 2 3  
>>> f(a=1)  
1 2 3
```

В случае передачи двух значений стандартное значение получит только аргумент c, а при передаче трех значений стандартные значения вообще не применяются:

```
>>> f(1, 4)      # Переопределение стандартных значений  
1 4 3  
>>> f(1, 4, 5)  
1 4 5
```

Наконец, ниже показано, как взаимодействуют ключевые аргументы и стандартные значения. Поскольку ключевые слова нарушают нормальное сопоставление по позиции слева направо, они по существу позволяют пропускать аргументы со стандартными значениями:

```
>>> f(1, c=6)          # Выбор стандартных значений
1 2 6
```

Здесь а получает значение 1 по позиции, с получает 6 по ключевому слову, а b – стандартное значение 2.

Будьте внимательны, чтобы не перепутать специальный синтаксис `имя=значение` в заголовке функции и в вызове функции; в *вызове* он означает ключевой аргумент, сопоставляемый по имени, тогда как в *заголовке* задает стандартное значение для необязательного аргумента. В обоих случаях это не оператор присваивания (несмотря на его внешний вид); он представляет собой специальный синтаксис для указанных двух контекстов, который видоизменяет механику сопоставления аргументов.

Комбинирование ключевых слов и стандартных значений

Далее приведен чуть больший пример, демонстрирующий в действии ключевые слова и стандартные значения. Вызывающий код должен всегда передавать, по крайней мере, два аргумента (для соответствия `spam` и `eggs`), но другие два аргумента необязательны. Если они опущены, то Python присваивает `toast` и `ham` стандартные значения, указанные в заголовке:

```
def func(spam, eggs, toast=0, ham=0):    # Первые два аргумента обязательны
    print((spam, eggs, toast, ham))

func(1, 2)                                # Вывод: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                     # Вывод: (1, 0, 0, 1)
func(spam=1, eggs=0)                      # Вывод: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)              # Вывод: (3, 2, 1, 0)
func(1, 2, 3, 4)                          # Вывод: (1, 2, 3, 4)
```

Снова обратите внимание, что в случае использования в вызове ключевых аргументов порядок, в котором перечисляются аргументы, несуществен; Python сопоставляет по имени, а не по позиции. В вызывающем коде должны быть предоставлены значения для `spam` и `eggs`, но они могут сопоставляться по позиции или по имени. Опять-таки имейте в виду, что форма `имя=значение` в вызове и в `def` имеет разный смысл – ключевой аргумент в вызове и стандартное значение в заголовке.



Остерегайтесь изменяемых стандартных значений. Как отмечалось в сноске в предыдущей главе, если вы укажете в качестве стандартного значения изменяемый объект (скажем, `def f(a=[])`), тогда *тот же самый* изменяемый объект будет повторно использоваться каждый раз, когда функция вызывается – даже в случае его модификации на месте внутри функции. В результате стандартное значение аргумента сохраняется из предыдущего вызова и не переустанавливается в первоначальное значение, закодированное в заголовке `def`. Чтобы переустанавливать его заново в каждом вызове, перенесите присваивание в тело функции. Изменяемые стандартные значения допускают сохранение состояния, но часто приводят к неожиданностям. Так как это настолько распространенная ловушка, мы отложим дальнейшие ее исследования до списка “затруднений” в конце главы 21.

Примеры произвольного количества аргументов

Последние два расширения при сопоставлении, * и **, предназначены для поддержки функций, которые принимают *любое количество* аргументов. Оба расширения могут встречаться либо в определении функции, либо в вызове функции, и в указанных двух местах они преследуют связанные цели.

Заголовки: сбор аргументов

Когда расширение `*` применяется в определении функции, оно обеспечивает сбор несопоставленных *позиционных* аргументов в кортеж:

```
>>> def f(*args): print(args)
```

При вызове такой функции Python собирает все позиционные аргументы в новый *кортеж* и присваивает его переменной `args`. Поскольку это нормальный объект кортежа, он допускает индексацию, проход в цикле `for` и т.д.:

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1, 2, 3, 4)
(1, 2, 3, 4)
```

Расширение `**` похоже, но только работает с *ключевыми* аргументами — оно собирает их в новый *словарь*, который затем можно обрабатывать с помощью обычных инструментов для словарей. В известном смысле форма `**` позволяет преобразовывать ключевые слова в словари, которые затем можно проходить посредством вызовов `keys`, словарных итераторов и т.п. (приблизительно так поступает вызов `dict` при передаче ему ключевых слов, но он *возвращает* новый словарь):

```
>>> def f(**args): print(args)
>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Наконец, в заголовках функций можно комбинировать нормальные аргументы с расширениями `*` и `**`, чтобы реализовывать очень гибкие сигнатуры вызовов. Например, в следующем взаимодействии значение 1 передается по позиции, 2 и 3 собираются в кортеж `pargs` с позиционными аргументами, а `x` и `y` попадают в словарь `kargs` с ключевыми аргументами:

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

Такой код встречается редко, но он обнаруживается в функциях, которым необходимо поддерживать множество шаблонов вызова (скажем, для обратной совместимости). На самом деле эти расширения допускается комбинировать даже более сложными способами, которые на первый взгляд могут показаться неоднозначными — мы вернемся к этой идеи позже в главе. Но сначала давайте посмотрим, что происходит, когда `*` и `**` присутствуют в вызовах функций, а не в их определениях.

Вызовы: распаковка аргументов

Во всех последних выпусках Python мы можем использовать синтаксис `*` также при вызове функции. В таком контексте данный синтаксис имеет смысл, противоположный его смыслу в определении функции — он распаковывает коллекцию аргументов, а не собирает ее. Например, мы можем передать функции четыре аргумента в кортеже и позволить Python распаковать их в индивидуальные аргументы:

```
>>> def func(a, b, c, d): print(a, b, c, d)
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)      # То же, что и func(1, 2, 3, 4)
1 2 3 4
```

Подобным же образом синтаксис `**` в вызове функции распаковывает словарь пар ключ/значение в отдельные ключевые аргументы:

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)      # То же, что и func(a=1, b=2, c=3, d=4)
1 2 3 4
```

И снова мы можем комбинировать нормальные, позиционные и ключевые аргументы очень гибкими способами:

```
>>> func(*[1, 2], **{'d': 4, 'c': 3})    # То же, что и func(1, 2, d=4, c=3)
1 2 3 4
>>> func(1, *(2, 3), **{'d': 4})        # То же, что и func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, c=3, *(2,), **{'d': 4})     # То же, что и func(1, 2, c=3, d=4)
1 2 3 4
>>> func(1, *(2, 3), d=4)                # То же, что и func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, *(2,), c=3, **{'d': 4})     # То же, что и func(1, 2, c=3, d=4)
1 2 3 4
```

Код такого рода удобен, когда спрогнозировать количество передаваемых функции аргументов на стадии написания сценария невозможно; замен во время выполнения строится коллекция аргументов и функция вызывается подобным способом. Опять-таки не путайте синтаксис аргументов с `/* */*` в заголовке функции и в вызове функции – в заголовке он собирает любое количество аргументов, а в вызове распаковывает любое количество аргументов. В обоих местах одна звездочка означает позиционные аргументы, тогда как две звездочки применяются к ключевым аргументам.



В главе 14 было показано, что форма `*args` в вызове является *итерационным контекстом* и потому формально она принимает любой итерируемый объект, а не только кортежи или другие последовательности, как демонстрировалось ранее в примерах. Скажем, если указать после `*` объект файла, то его строки распакуются в индивидуальные аргументы (например, `func(*open('fname'))`). В главе 20 после исследования генераторов будут приведены дополнительные примеры.

Такая универсальность поддерживается в Python 3.X и 2.X, но только для *вызовов* – аргумент `*args` в вызове допускает любой итерируемый объект, в то время как та же самая форма в заголовке `def` всегда объединяет добавочные аргументы в *кортеж*. Описанное поведение заголовка по духу и синтаксису похоже на поведение `*` в расширенных формах распаковывающего присваивания последовательностей из Python 3.X, которые встречались в главе 11 (вроде `x, *y = z`), хотя использование звездочки всегда приводит к созданию списков, а не кортежей.

Обобщенное применение функций

Примеры в предыдущем разделе могли показаться учебными (если вообще не экзотическими), но они используются чаще, чем возможно ожидалось. Некоторые программы нуждаются в вызове произвольных функций в обобщенной манере, не зная заранее их имена или аргументы. Фактически реальная мощь специального синтаксиса вызовов с переменным количеством аргументов заключается в том, что при написании сценария вы не знаете, сколько аргументов требует вызов функции. Скажем, вы можете применять логику `if` для выбора из набора функций и списков аргументов и вызывать любую из них обобщенным образом (функции в ряде следующих далее примеров являются гипотетическими):

```
if sometest:
    action, args = func1, (1,)                      # Вызов func1 с одним аргументом
else:
    action, args = func2, (1, 2, 3)                  # Вызов func2 с тремя аргументами
...и так далее...
action(*args)                                       # Выполнение обобщенным образом
```

Здесь задействована форма `*` и тот факт, что функции представляют собой объекты, на которые можно ссылаться и вызывать через любую переменную. В более общем случае такой синтаксис вызовов с переменным количеством аргументов удобен всякий раз, когда предугадать список аргументов невозможно. Например, если пользователь выбирает произвольную функцию через какой-то пользовательский интерфейс, тогда у вас может отсутствовать возможность жестко закодировать вызов функции при написании сценария. В качестве обходного способа просто постройте список аргументов с помощью операций над последовательностями и вызовите функцию с использованием синтаксиса аргумента со звездочкой, чтобы распаковать аргументы:

```
>>> ... определить либо импортировать func3...
>>> args = (2, 3)
>>> args += (4,)
>>> args
(2, 3, 4)
>>> func3(*args)
```

Поскольку список аргументов здесь передается как кортеж, программа может построить его во время выполнения. Эта методика также оказывается полезной для функций, которые тестируют либо измеряют время работы других функций. Скажем, в приведенном далее коде мы поддерживаем любую функцию с любыми аргументами, передавая ей все, что было отправлено (файл `tracer0.py` в пакете примеров для книги):

```
def tracer(func, *pargs, **kargs):      # Принимает произвольные аргументы
    print('calling:', func.__name__)
    return func(*pargs, **kargs)          # Передает произвольные аргументы

def func(a, b, c, d):
    return a + b + c + d
    print(tracer(func, 1, 2, c=3, d=4))
```

В коде применяется встроенный атрибут `__name__`, присоединяемый к каждой функции (он вполне ожидаемо содержит строку с именем функции), и синтаксис со звездочками для сбора и последующей распаковки аргументов, предназначенных для обработки функцией. Другими словами, когда код выполняется, аргументы перехватываются функцией `tracer` и затем передаются посредством синтаксиса вызовов с переменным количеством аргументов:

```
calling: func  
10
```

Еще один пример использования такой методики предлагался во врезке “Что потребует внимания: настройка `open`” в конце предыдущей главы, где она применялась для переустановки встроенной функции `open`. Позже в книге мы еще рассмотрим дополнительные примеры использования данного приема, в частности при измерении времени в главе 21 и при написании разнообразных декораторных утилит в главе 39. Она представляет собой распространенную технику в рамках универсальных инструментов.

Исчезнувшая встроенная функция `apply` (Python 2.X)

До выхода Python 3.X эффекта синтаксиса вызовов с переменным количеством аргументов `*аргументы` и `**аргументы` можно было достичь с помощью встроенной функции по имени `apply`. Эта первоначальная методика была удалена в Python 3.X ввиду своей избыточности (в Python 3.X избавились от многих покрытых пылью инструментов, по большому счету бесцельно существовавших годами). Тем не менее, она все еще доступна во всех выпусках Python 2.X и ее можно встретить в более старом коде Python 2.X.

Вот эквивалентный код, применяемый до выхода Python 3.X:

```
func(*pargs, **kargs) # Более новый синтаксис вызовов:  
функция(*последовательность, **словарь)  
apply(func, pargs, kargs) # Исчезнувшая встроенная функция:  
apply(функция, последовательность, словарь)
```

Например, рассмотрим следующую функцию, которая принимает любое количество позиционных или ключевых аргументов:

```
>>> def echo(*args, **kwargs): print(args, kwargs)  
>>> echo(1, 2, a=3, b=4)  
(1, 2) {'a': 3, 'b': 4}
```

В Python 2.X мы можем вызывать ее обобщенным образом, используя `apply` или синтаксис, который теперь обязателен в Python 3.X:

```
>>> pargs = (1, 2)  
>>> kargs = {'a':3, 'b':4}  
>>> apply(echo, pargs, kargs)  
(1, 2) {'a': 3, 'b': 4}  
>>> echo(*pargs, **kargs)  
(1, 2) {'a': 3, 'b': 4}
```

Обе формы также работают для встроенных функций в Python 2.X (обратите внимание на наличие хвостовой буквы `L` для длинных целых Python 2.X):

```
>>> apply(pow, (2, 100))  
1267650600228229401496703205376L  
>>> pow(*(2, 100))  
1267650600228229401496703205376L
```

Форма синтаксиса распаковывающих вызовов новее функции `apply`, она в целом предпочтительнее и является обязательной в Python 3.X. (Формально этот синтаксис был добавлен в версии Python 2.0; функция `apply` объявлена устаревшей в Python 2.3, по-прежнему может применяться без выдачи предупреждений в Python 2.7, но исчезла в версии Python 3.X.) Помимо симметрии с формами сбора аргументов `*` в заголов-

ках `def` и того факта, что он требует меньшего клавиатурного набора, более новый синтаксис вызовов также позволяет передавать дополнительные аргументы, не требуя ручного расширения последовательностей или словарей:

```
>>> echo(0, c=5, *pargs, **kargs) # Нормальный аргумент, ключевой аргумент,
#      *последовательность, **словарь
(0, 1, 2) {'a': 3, 'c': 5, 'b': 4}
```

То есть такая форма синтаксиса вызовов *более универсальна*. Так как она обязательна в Python 3.X, вы должны полностью забыть о функции `apply` (конечно, если только она не встречается в коде Python 2.X, который вам приходится сопровождать...).

Аргументы с передачей только по ключевым словам Python 3.X

В Python 3.X обобщены правила упорядочения в заголовках функций, чтобы позволить указывать *аргументы с передачей только по ключевым словам* – аргументы, которые должны передаваться только по ключевому слову и никогда не будут заполняться позиционными аргументами. Они полезны, когда необходима функция, которая и обрабатывает любое количество аргументов, и принимает (возможно, необязательные) конфигурационные параметры.

Синтаксически аргументы с передачей только по ключевым словам записываются как именованные аргументы, которые могут появляться после конструкции `*аргументы` в списке аргументов. Значения для таких аргументов должны передаваться в вызове с использованием синтаксиса ключевых аргументов. Например, в следующем коде Python 3.X аргумент `a` может передаваться по имени или по позиции, `b` собирает добавочные позиционные аргументы, а `c` должно передаваться только по ключевому слову:

```
>>> def kwonly(a, *b, c):
    print(a, b, c)

>>> kwonly(1, 2, c=3)
1 (2,) 3
>>> kwonly(a=1, c=3)
1 () 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
Ошибка типа: kwonly() отсутствует 1 обязательный аргумент с передачей только по ключевым словам: c
```

Мы также можем применять в списке аргументов сам по себе символ `*`, указывая на то, что функция не принимает список аргументов переменной длины, но по-прежнему ожидает передачи всех аргументов, следующих за символом `*`, по ключевому слову. В показанной ниже функции аргумент `a` снова может передаваться по позиции или по имени, но аргументы `b` и `c` обязаны передаваться по ключевому слову, и никакие добавочные позиционные аргументы не разрешены:

```
>>> def kwonly(a, *, b, c):
    print(a, b, c)

>>> kwonly(1, c=3, b=2)
1 2 3
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2, 3)
```

```
TypeError: kwonly() takes 1 positional argument but 3 were given
Ошибка типа: kwonly() принимает 1 позиционный аргумент, но было передано 3
>>> kwonly(1)
TypeError: kwonly() missing 2 required keyword-only arguments: 'b' and 'c'
Ошибка типа: kwonly() отсутствуют 2 обязательных аргумента с передачей
только по ключевым словам: b и c
```

Вы по-прежнему можете использовать стандартные значения для аргументов с передачей только по ключевым словам, хотя они находятся после символа * в заголовке функции. В следующем коде а может передаваться по имени или по позиции, а b и c являются необязательными, но должны передаваться по ключевым словам, если присутствуют:

```
>>> def kwonly(a, *, b='spam', c='ham'):
    print(a, b, c)

>>> kwonly(1)
1 spam ham
>>> kwonly(1, c=3)
1 spam 3
>>> kwonly(a=1)
1 spam ham
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given
Ошибка типа: kwonly() принимает 1 позиционный аргумент, но было передано 2
```

На самом деле аргументы с передачей только по ключевым словам со стандартными значениями необязательны, но такие аргументы без стандартных значений становятся обязательными *ключевыми словами* для функции:

```
>>> def kwonly(a, *, b, c='spam'):
    print(a, b, c)

>>> kwonly(1, b='eggs')
1 eggs spam
>>> kwonly(1, c='eggs')
TypeError: kwonly() missing 1 required keyword-only argument: 'b'
Ошибка типа: kwonly() отсутствует 1 обязательный аргумент с передачей только
по ключевым словам: b
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given
Ошибка типа: kwonly() принимает 1 позиционный аргумент, но было передано 2
>>> def kwonly(a, *, b=1, c, d=2):
    print(a, b, c, d)

>>> kwonly(3, c=4)
3 1 4 2
>>> kwonly(3, c=4, b=5)
3 5 4 2
>>> kwonly(3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
Ошибка типа: kwonly() отсутствует 1 обязательный аргумент с передачей только
по ключевым словам: c
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given
Ошибка типа: kwonly() принимает 1 позиционный аргумент, но было передано 3
```

Правила упорядочения

Обратите внимание, что аргументы с передачей только по ключевым словам должны указываться после одиночной звездочки, а не двух — именованные аргументы не могут находиться после формы с произвольным количеством ключевых аргументов ****аргументы**, а конструкция ****** не может появляться сама по себе в списке аргументов. Следующие две попытки приводят к синтаксической ошибке:

```
>>> def kwonly(a, **pargs, b, c):
SyntaxError: invalid syntax
Синтаксическая ошибка: недопустимый синтаксис
>>> def kwonly(a, **, b, c):
SyntaxError: invalid syntax
Синтаксическая ошибка: недопустимый синтаксис
```

Это означает, что в *заголовке* функции аргументы с передачей только по ключевым словам должны записываться перед формой с произвольным количеством ключевых аргументов ****аргументы** и после формы с произвольным количеством позиционных аргументов ****аргументы**, когда они обе присутствуют. Всякий раз, когда имя аргумента появляется перед конструкцией ****аргументы**, он может быть позиционным аргументом со стандартным значением, а не аргументом с передачей только по ключевым словам:

```
>>> def f(a, *b, **d): print(a, b, c, d) # Перед ** допускаются аргументы
                                                # с передачей только по ключевым словам!
SyntaxError: invalid syntax
Синтаксическая ошибка: недопустимый синтаксис
>>> def f(a, *b, c=6, **d): print(a, b, c, d) # Сбор аргументов в заголовке
>>> f(1, 2, 3, x=4, y=5)                      # Используется стандартное значение
1 (2, 3) 6 {'y': 5, 'x': 4}
>>> f(1, 2, 3, x=4, y=5, c=7)                 # Переопределяется стандартное значение
1 (2, 3) 7 {'y': 5, 'x': 4}
>>> f(1, 2, 3, c=7, x=4, y=5)                 # Где угодно в ключевых аргументах
1 (2, 3) 7 {'y': 5, 'x': 4}
>>> def f(a, c=6, *b, **d): print(a, b, c, d)      # Здесь c - не аргумент
                                                # с передачей только по ключевым словам!
>>> f(1, 2, 3, x=4)
1 (3,) 2 {'x': 4}
```

Фактически в *вызовах* функций сохраняют справедливость похожие правила упорядочения: когда передаются аргументы с передачей только по ключевым словам, они обязаны находиться перед формой ****аргументы**. Однако аргумент с передачей только по ключевым словам может быть указан либо перед, либо после формы ****аргументы** и также включен в нее:

```
>>> def f(a, *b, c=6, **d): print(a, b, c, d) # Аргумент с передачей только
                                                # по ключевым словам между * и **
>>> f(1, *(2, 3), **dict(x=4, y=5))        # Распаковка аргументов при вызове
1 (2, 3) 6 {'y': 5, 'x': 4}
>>> f(1, *(2, 3), **dict(x=4, y=5), c=7)    # Ключевые аргументы перед формой **аргументы!
SyntaxError: invalid syntax
Синтаксическая ошибка: недопустимый синтаксис
>>> f(1, *(2, 3), c=7, **dict(x=4, y=5))    # Переопределение стандартного значения
1 (2, 3) 7 {'y': 5, 'x': 4}
```

```
>>> f(1, c=7, *(2, 3), **dict(x=4, y=5))      # После или перед *
1 (2, 3) 7 {'y': 5, 'x': 4}
>>> f(1, *(2, 3), **dict(x=4, y=5, c=7))      # Аргумент с передачей только
                                                # по ключевым словам в **
1 (2, 3) 7 {'y': 5, 'x': 4}
```

Отследите представленные сценарии самостоятельно в сочетании с формально описанными ранее общими правилами упорядочения аргументов. В приведенных здесь искусственных примерах они могут показаться наихудшими случаями, но на практике с ними приходится сталкиваться, в особенности тем, кто разрабатывает библиотеки и инструменты для применения другими программистами на Python.

Для чего используются аргументы с передачей только по ключевым словам?

Итак, когда нас могут интересовать аргументы с передачей только по ключевым словам? Выражаясь кратко, они облегчают для функции возможность принимать любое количество позиционных аргументов, подлежащих обработке, и конфигурационные параметры, передаваемые как ключевые аргументы. Хотя применять их необязательно, без аргументов с передачей только по ключевым словам может требоваться дополнительная работа, связанная с предоставлением стандартных значений для таких параметров и контролем того, что избыточные ключевые аргументы не передавались.

Представим себе функцию, которая обрабатывает набор передаваемых объектов и позволяет передавать флаг отслеживания:

```
process(X, Y, Z)                      # Использование стандартного значения флага
process(X, Y, notify=True)             # Переопределение стандартного значения флага
```

В отсутствие аргументов с передачей только по ключевым словам нам пришлось бы использовать формы `*аргументы` и `**аргументы` и вручную инспектировать ключевые слова, но применение аргументов с передачей только по ключевым словам требует написания меньшего объема кода. Следующий заголовок `def` гарантирует, что никакой позиционный аргумент не будет некорректно сопоставлен с `notify`, и в случае его передачи требует, чтобы он был ключевым:

```
def process(*args, notify=False): ...
```

В разделе “Эмуляция функции `print` из Python 3.X” далее в главе мы рассмотрим более реалистичный пример. Демонстрация аргументов с передачей только по ключевым словам в действии представлена в учебном примере, посвященном измерению времени альтернативных версий итерации, в главе 21. Дополнительные усовершенствования определений функций в Python 3.X обсуждаются при описании синтаксиса аннотирования функций в главе 19.

ФУНКЦИЯ `min`

Итак, пришло время заняться чем-то более реалистичным. Давайте проработаем упражнение, которое продемонстрирует практическое приложение инструментов со-поставления аргументов.

Предположим, что необходимо написать функцию, которая способна находить минимальное значение в произвольном наборе аргументов и произвольном наборе типов данных. То есть функция обязана принимать ноль и более аргументов – сколько, сколько передается.

Кроме того, функция должна работать со всеми типами объектов Python: числами, строками, списками, списками словарей, файлами и даже `None`. (Справедливости ради

следует отметить, что пользователям Python 3.X не нужно поддерживать словари, потому что их словари не поддерживают прямые сравнения; см. главы 8 и 9.)

Первое требование предоставляет собой естественный пример того, как можно эффективно задействовать средство * — мы можем сбрать аргументы в кортеж и пройти по ним с помощью простого цикла `for`. Вторая часть постановки задачи сложностью не отличается: из-за того, что каждый тип объекта поддерживает сравнения, нам не придется специализировать функцию по типам (случай употребления *полиморфизма*); мы просто вслепую сравниваем объекты и позволяем Python выяснить то, сравнение какого вида выполнять в отношении сравниваемых объектов.

Основная задача

В файле `mins.py` показаны три способа реализации данной операции, по крайней мере, один из которых был предложен моим студентом (я часто предлагаю группе студентов решить такое упражнение, чтобы вывести их из полусонного состояния после обеда).

- Первая функция извлекает первый аргумент (`args` является кортежем) и проходит по остальным путем нарезания первого (нет смысла сравнивать объект с самим собой, особенно если он может быть крупной структурой).
- Вторая функция позволяет Python автоматически выбирать первый и остальные аргументы, тем самым избегая индексирования и нарезания.
- Третья функция преобразует кортеж в список посредством встроенного вызова `list` и задействует списковый метод `sort`.

Метод `sort` написан на С, так что временами он может оказываться быстрее других подходов, но линейный просмотр в первых двух версиях способен делать их быстрее в большинстве случаев¹. Вот три решения из файла `mins.py`:

```
def min1(*args):  
    res = args[0]  
    for arg in args[1:]:  
        if arg < res:  
            res = arg  
    return res  
  
def min2(first, *rest):  
    for arg in rest:  
        if arg < first:  
            first = arg  
    return first
```

¹ На самом деле все довольно сложно. Процедура `sort` в Python написана на С и использует крайне оптимизированный алгоритм, который старается извлечь преимущество из частичного упорядочения элементов, подлежащих сортировке. Он назван `timsort` в честь своего создателя Тима Петерса (Tim Peters) и в его документации заявлено, что временами алгоритм обладает “сверхъестественной производительностью” (совсем неплохо как для сортировки!). Тем не менее, сортировка по своей сути является быстрорастущей операцией (она обязана разбивать последовательность и собирать ее снова много раз), а другие версии просто выполняют один линейный просмотр слева направо. Совокупный эффект в том, что сортировка проходит быстрее, если аргументы частично упорядочены, но в противном случае вероятно будет медленнее (что подтверждается тестовыми запусками в Python 3.X). Но даже при этих условиях производительность Python со временем может меняться, и тот факт, что сортировка реализована на языке С, способен значительно помочь; для точного анализа потребуется измерить время выполнения альтернативных версий посредством модуля `time` или `timeit`, как будет показано в главе 21.

```
def min3(*args):
    tmp = list(args)      # Или в Python 2.4+: return sorted(args)[0]
    tmp.sort()
    return tmp[0]

print(min1(3, 4, 1, 2))
print(min2("bb", "aa"))
print(min3([2,2], [1,1], [3,3]))
```

Все три решения после запуска файла производят тот же самый результат. Попробуйте самостоятельно поэкспериментировать, набрав несколько вызовов в интерактивной подсказке:

```
% python mins.py
1
aa
[1, 1]
```

Обратите внимание, что ни в одной из трех версий не проверяется случай, когда аргументы вообще не передаются. Такую проверку можно было бы предусмотреть, но смысла в ней мало — во всех трех решениях Python автоматически сгенерирует исключение, если не было передано ни одного аргумента. В первой версии исключение генерируется при попытке извлечь элемент 0, во второй — когда Python обнаружит несоответствие списку аргументов, а в третьей — при попытке возвратить элемент 0 в конце.

Поведение является именно тем, что нам нужно — поскольку написанные функции поддерживают любой тип данных, отсутствует какое-то сигнальное значение, которое мы могли бы передать обратно, указав на ошибку, так что мы можем в равной степени разрешить генерацию исключения. Существуют отклонения от этого правила (например, вы можете самостоятельно проверять на предмет ошибок, если предпочитаете избегать действий, выполняемых до достижения кода, который автоматически инициирует ошибку), но в целом лучше допускать, что аргументы будут работать в коде ваших функций, и позволить Python генерировать ошибки в противном случае.

Дополнительные очки

Здесь вы можете заработать дополнительные очки, изменив функции так, чтобы они находили *максимальные* значения, а не минимальные. Работа несложная: первые две версии требуют лишь изменения < на >, а в третьей просто понадобится возвращать `tmp[-1]` вместо `tmp[0]`. Чтобы получить еще больше очков, не забудьте поменять имя функции на `max` (хотя это совершенно необязательно).

Можно также обобщить единственную функцию для нахождения минимального или максимального значения путем оценки строк с выражениями сравнения с помощью инструмента вроде встроенной функции `eval` (см. руководство по библиотеке либо разнообразные примеры ее использования в книге, особенно в главе 10) или за счет передачи произвольной функции сравнения. Последняя схема реализована в файле `minmax.py`:

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res
```

```

def lessthan(x, y): return x < y           # См. также: lambda, eval
def grtrthan(x, y): return x > y
print(minmax(lessthan, 4, 2, 1, 5, 6, 3))    # Тестовый код
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
% python minmax.py
1
6

```

Функции являются еще одним видом объекта, который можно передавать в функцию, подобную показанной. Скажем, чтобы сделать ее функцией `max` (или другой), мы просто передаем функцию `test` надлежащего вида. Может показаться, что мы вынуждены делать дополнительную работу, но основной смысл такого обобщения функций — вместо вырезания и вставки с целью изменения всего лишь одного символа — заключается в том, что у нас остается только одна версия для изменения в будущем, а не две.

Заключение

Разумеется, все это было обычным упражнением по написанию кода. В действительности нет никаких причин писать код для функции `min` или `max`, потому что они доступны как встроенные функции в Python! Мы бегло сталкивались с ними в главе 5 при обсуждении инструментов для обработки чисел и еще раз в главе 14, когда исследовали итерационные контексты. Встроенные версии работают в точности как наши, но они реализованы на языке C для обеспечения оптимальной скорости и принимают либо одиночный итерируемый объект, либо множество аргументов. Однако, несмотря на избыточность в данном контексте, применяемый здесь универсальный кодовый шаблон может оказаться удобным в других сценариях.

Обобщенные функции для работы с множествами

Давайте рассмотрим более полезный пример со специальными режимами сопоставления аргументов. В конце главы 16 мы написали функцию, которая возвращала пересечение двух последовательностей (она выбирала элементы, присутствующие в обеих последовательностях). Ниже представлена версия, которая ищет пересечение произвольного числа последовательностей (одной и более), используя форму сопоставления с переменным количеством аргументов *аргументы для сбора всех переданных аргументов. Поскольку аргументы поступают в виде кортежа, мы можем обрабатывать произвольное число аргументов с целью сбора элементов, имеющихся во всех операндах:

```

def intersect(*args):
    res = []
    for x in args[0]:
        # Просмотр первой последовательности
        if x in res: continue
        for other in args[1:]:
            # Пропуск дубликатов
            if x not in other: break
            # Для всех остальных аргументов
            # Элемент находится во всех
            # последовательностях?
            else:
                # Нет: выйти из цикла
                res.append(x)
                # Да: добавить элементы в конец
    return res

```

```

def union(*args):
    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 res

```

Из-за того, что такие инструменты заслуживают многократного применения (и слишком объемные, чтобы повторно набирать их код в интерактивной подсказке), мы сохраним функции в файле модуля по имени `inter2.py` (если вы забыли, как работают модули и импортирование, тогда освежите в памяти введение в главе 3 или дождитесь полного изложения в части V). В обеих функциях аргументы, переданные при вызове, поступают в виде кортежа `args`. Подобно первоначальной функции `intersect` обе работают на последовательностях любого типа. Вот как функции обрабатывают строки, разнородные типы и более двух последовательностей:

```

% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"
>>> intersect(s1, s2), union(s1, s2)           # Два операнда
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])
>>> intersect([1, 2, 3], (1, 4))             # Разнородные типы
[1]
>>> intersect(s1, s2, s3)                   # Три операнда
['S', 'A', 'M']
>>> union(s1, s2, s3)
['S', 'P', 'A', 'M', 'C', 'L']

```

Для более основательного тестирования мы написали функцию для применения двух инструментов к аргументам в разном порядке, используя простую методику тасования, которая встречалась в главе 13. На каждой итерации цикла она посредством нарезания перемещает первый элемент в конец, применяет `*` для распаковки аргументов и выполняет сортировку, чтобы результаты можно было сравнивать:

```

>>> def tester(func, items, trace=True):
    for i in range(len(items)):
        items = items[1:] + items[:1]
        if trace: print(items)
        print(sorted(func(*items)))

>>> tester(intersect, ('a', 'abcdefg', 'abdst', 'albmnd'))
('abcdefg', 'abdst', 'albmnd', 'a')
['a']
('abdst', 'albmnd', 'a', 'abcdefg')
['a']
('albmnd', 'a', 'abcdefg', 'abdst')
['a']
('a', 'abcdefg', 'abdst', 'albmnd')
['a']

>>> tester(union, ('a', 'abcdefg', 'abdst', 'albmnd'), False)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']

```

```
>>> tester(intersect, ('ba', 'abcdefg', 'abdst', 'albmnd'), False)
[['a', 'b']]
[['a', 'b']]
[['a', 'b']]
[['a', 'b']]
```

Перестановка аргументов здесь не обеспечивает генерацию всех возможных порядков следования аргументов (для этого потребовалась бы полная перестановка — 24 порядка для 4 аргументов), но их достаточно для выяснения, влияет ли порядок аргументов на результаты. Продолжив тестирование, вы заметите, что дубликаты не появляются ни в результатах пересечения, ни в результатах объединения, что с точки зрения математики квалифицирует их как операции над множествами:

```
>>> intersect([1, 2, 1, 3], (1, 1, 4))
[1]
>>> union([1, 2, 1, 3], (1, 1, 4))
[1, 2, 3, 4]
>>> tester(intersect, ('ababa', 'abcdefga', 'aaaab'), False)
[['a', 'b']]
[['a', 'b']]
[['a', 'b']]
```

С алгоритмической точки зрения они все еще далеки от оптимальных, но с учетом замечания в следующей далее врезке “На заметку!” мы оставляем дальнейшее совершенствование кода в качестве упражнения для самостоятельного решения. Также имейте в виду, что перестановка аргументов в функции `tester` может оказаться полезным инструментом в общем смысле, и функция `tester` стала бы проще, если делигировать выполнение перестановки другой функции, которая по своему усмотрению могла бы создавать или генерировать комбинации аргументов:

```
>>> def tester(func, items, trace=True):
    for args in scramble(items):
        ...использовать аргументы...
```

На самом деле мы займемся этим — пример будет соответствующим образом переработан в главе 20, когда мы выясним, каким образом писать определяемые пользователем *генераторы*. Кроме того, в главе 32 мы перепишем код операций над множествами еще раз, а решение упражнения для части VI в виде *классов* расширит объект списка методами.



Поскольку Python теперь располагает *типовом множества* (описанным в главе 5), ни один из примеров обработки множеств в книге больше не требуется; они включены только для демонстрации методик написания кода и в наши дни являются всего лишь учебными. За счет постоянного улучшения и расширения, похоже, Python устраивает коварный заговор против моих примеров в книге, делая их устаревшими!

Эмуляция функции `print` из Python 3.X

В завершение главы давайте рассмотрим последний пример сопоставления аргументов в работе. Показанный далее код предназначен для использования в Python 2.X и более ранних версиях (он работает также в Python 3.X, но в нем нет никакого смысла). В коде применяются кортеж с произвольным количеством позиционных аргументов `*аргументы` и словарь с произвольным количеством ключевых аргументов `**аргументы`:

тов `**аргументы` для эмуляции большинства того, что делает функция `print` из Python 3.X. Вообще говоря, в Python 3.X могли бы предложить код подобного рода как *вариант*, не удаляя `print` из Python 2.X, но взамен было решено полностью порвать с прошлым.

Как объяснялось в главе 11, в действительности это не требуется, потому что программисты на Python 2.X всегда могут включить функцию `print` из Python 3.X с помощью импортирования в следующей форме (доступного в версиях Python 2.6 и 2.7):

```
from __future__ import print_function
```

Тем не менее, чтобы продемонстрировать сопоставление аргументов в целом, в файле `print3.py`, содержимое которого приведено ниже, делается та же самая работа в небольшой порции многократно используемого кода путем построения выводимой строки с учетом конфигурационных аргументов:

```
#!python
"""
Эмулирует большую часть функции print из Python 3.X для применения в Python
2.X (и Python 3.X).
Сигнатура вызова: print3(*args, sep=' ', end='\n', file=sys.stdout)
"""

import sys

def print3(*args, **kargs):
    sep = kargs.get('sep', ' ') # Keyword arg defaults
    end = kargs.get('end', '\n')
    file = kargs.get('file', sys.stdout)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

Чтобы протестировать функцию `print3`, ее необходимо импортировать в другом файле или в интерактивной подсказке и применять подобно функции `print` из Python 3.X. Вот содержимое тестового сценария `testprint3.py` (обратите внимание, что функция должна называться `print3`, потому что `print` – зарезервированное слово в Python 2.X):

```
from print3 import print3
print3(1, 2, 3)
print3(1, 2, 3, sep='')           # Подавление вывода разделителя
print3(1, 2, 3, sep='...')
print3(1, [2], (3,), sep='...')   # Разнообразные типы объектов
print3(4, 5, 6, sep='', end='')   # Подавление вывода символа новой строки
print3(7, 8, 9)
print3()                         # Добавление символа новой строки (или пустой строки)
import sys
print3(1, 2, 3, sep='??', end='.\n', file=sys.stderr) # Перенаправление в файл
```

Запустив `testprint3.py` под управлением Python 2.X, мы получаем те же результаты, которые дает функция `print` из Python 3.X:

```
C:\code> c:\python27\python testprint3.py
1 2 3
123
```

```
1...2...3
1...[2]...(3,)
4567 8 9
1?????3.
```

При выполнении в Python 3.X, хотя это не имеет особого смысла, результаты будут идентичными. Как обычно, универсальность структуры Python позволяет нам создавать прототипы или развивать концепции в самом языке Python. В данном случае инструменты сопоставления аргументов в коде Python оказываются в той же степени гибкими, как и во внутренней реализации Python.

Использование аргументов с передачей только по ключевым словам

Интересно отметить, что рассмотренный выше пример можно было бы реализовать с применением аргументов с передачей только по ключевым словам Python 3.X, описанных ранее в главе, для автоматической проверки конфигурационных аргументов. Вот вариант кода из файла print3_alt1.py:

```
#!python3
"Использование аргументов с передачей только по ключевым словам Python 3.X"
import sys

def print3(*args, sep=' ', end='\n', file=sys.stdout):
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

Версия работает точно так же, как оригинал, и является главным примером, насколько полезными могут оказаться аргументы с передачей только по ключевым словам. В первоначальной версии предполагается, что все позиционные аргументы подлежат выводу, а все ключевые аргументы предназначены только для конфигурационных параметров. Этого почти достаточно, но любые добавочные ключевые аргументы молча игнорируются. Вызов вроде следующего, например, корректно генерирует исключение, когда функция имеет аргументы с передачей только по ключевым словам:

```
>>> print3(99, name='bob')
TypeError: print3() got an unexpected keyword argument 'name'
Ошибка типа: print3() получила непредвиденный ключевой аргумент name
```

но молча проигнорирует аргумент name в первоначальной версии функции. Для обнаружения избыточных ключевых аргументов вручную мы могли бы использовать dict.pop(), чтобы удалять извлеченные элементы и проверять, не пуст ли словарь. Показанная далее версия из файла print3_alt2.py эквивалентна версии, где применяются аргументы с передачей только по ключевым словам — она генерирует встроенное исключение с помощью оператора raise, который работает так, как если бы исключение генерировал Python (мы исследуем его более детально в части VII):

```
#!python
"Использование удаления ключевых аргументов Python 2.X/3.X со стандартными
значениями"
import sys
```

```
def print3(*args, **kargs):
    sep = kargs.pop('sep', ' ')
    end = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    if kargs: raise TypeError('extra keywords: %s' % kargs)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

Новая версия `print3` работает, как и ранее, но теперь также перехватывает избыточные ключевые аргументы:

```
>>> print3(99, name='bob')
TypeError: extra keywords: {'name': 'bob'}
Ошибка типа: добавочные ключевые аргументы: {'name': 'bob'}
```

Данная версия функции выполняется под управлением Python 2.X, но требует на четыре строки больше кода, чем версия, использующая аргументы с передачей только по ключевым словам. К сожалению, в этом случае дополнительный код неизбежен — версия, где применяются аргументы с передачей только по ключевым словам, работает лишь в Python 3.X, что сводит на нет большинство причин, по которым вообще был написан пример: эмулятор функции Python 3.X, работающий только в Python 3.X, не особенно полезен! Однако в программах, написанных для запуска только в Python 3.X, аргументы с передачей только по ключевым словам могут упростить особую категорию функций, которые принимают и аргументы, и конфигурационные параметры. Еще один случай использования аргументов с передачей только по ключевым словам ищите в учебном примере измерения времени выполнения для альтернативных версий итерации в главе 21.

Что потребует внимания: ключевые аргументы

Как вы возможно уже заметили, расширенные режимы сопоставления атрибутов могут быть сложными. Они также в основном необязательны в коде; вы можете обойтись простым сопоставлением по позиции, и вероятно имеет смысл так поступать, когда вы только начинаете. Тем не менее, поскольку некоторые инструменты Python их задействуют, важно иметь общее представление об этих режимах.

Например, ключевые аргументы играют важную роль в модуле `tkinter`, де-факто стандартном API-интерфейсе для создания графических пользовательских интерфейсов на Python (в Python 2.X он называется `Tkinter`). Мы кратко упоминали `tkinter` в нескольких местах книги, но что касается его шаблона вызовов, то ключевые аргументы устанавливают конфигурационные параметры при построении компонентов графического пользовательского интерфейса. Скажем, вызов в форме:

```
from tkinter import *
widget = Button(text="Press me", command=someFunction)
```

создает новую кнопку, а также указывает ее текст и функцию обратного вызова с применением ключевых аргументов `text` и `command`. Поскольку количество конфигурационных параметров для виджета может быть большим, ключевые аргументы позволяют выбирать интересующие параметры. Без них, возможно, пришлось бы либо перечислить все возможные параметры по позиции, либо надеяться на то, что разумный протокол для стандартных значений аргументов обработает все вероятные композиции параметров.

Многие встроенные функции в Python также ожидают использования ключевых аргументов для своих рабочих параметров, которые могут иметь или не иметь стандартные значения. Например, как было показано в главе 8, встроенная функция `sorted`:

```
sorted(iterable, key=None, reverse=False)
```

ожидает передачи итерируемого объекта, подлежащего сортировке, но также позволяет передавать необязательные ключевые параметры, чтобы указывать функцию сортировки словарных ключей и флаг изменения направления на противоположное, для которых предусмотрены стандартные значения `None` и `False` соответственно. Так как эти параметры мы обычно не применяем, их можно опустить и использовать стандартные значения.

Как упоминалось ранее, вызовы `dict`, `str.format` и `print` из Python 3.X также принимают ключевые параметры – другие способы применения, которые нам приходилось представлять в предшествующих главах из-за их прямой зависимости от исследуемых здесь режимов передачи аргументов (увы, те, кто изменяют Python, уже знают Python!).

Резюме

В главе обсуждалась вторая из двух основных концепций, связанных с функциями – *аргументы*, которые являются способом передачи объектов функции. Вы узнали, что аргументы передаются функции по присваиванию, т.е. по ссылкам на объекты (в действительности по указателям). Кроме того, исследовались более сложные расширения, в том числе стандартные и ключевые аргументы, инструменты для использования произвольно большого количества аргументов и аргументы с передачей только по ключевым словам в Python 3.X. Наконец, было показано, что изменяемые аргументы могут демонстрировать такое же поведение, как и другие разделяемые ссылки на объекты – если объект явно не скопировать при передаче, то модификация переданного изменяемого объекта внутри функции может повлиять на вызывающий код.

В следующей главе мы продолжим рассмотрение функций исследованием ряда более развитых идей, связанных с функциями, среди которых аннотации функций, рекурсия, выражения `lambda` и функциональные инструменты наподобие `map` и `filter`. Многие концепции происходят из того факта, что функции являются нормальными объектами в Python, а потому поддерживают расширенные и очень гибкие режимы обработки. Но прежде чем двигаться дальше, закрепите пройденный материал этой главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

В большинстве вопросов результаты могут слегка отличаться в Python 2.X – быть заключенными в круглые скобки и содержать запятые при выводе множественных значений. Чтобы результаты в Python 2.X точно соответствовали результатам, приведенным для Python 3.X, первым делом импортируйте `print_function` из `__future__`.

1. Каким будет вывод следующего кода и почему?

```
>>> def func(a, b=4, c=5):
    print(a, b, c)

>>> func(1, 2)
```

2. Каким будет вывод такого кода и почему?

```
>>> def func(a, b, c=5):
    print(a, b, c)
>>> func(1, c=3, b=2)
```

3. Что насчит следующего кода: каким будет вывод и почему?

```
>>> def func(a, *pargs):
    print(a, pargs)
>>> func(1, 2, 3)
```

4. Что выведет этот код и почему?

```
>>> def func(a, **kargs):
    print(a, kargs)
>>> func(a=1, c=3, b=2)
```

5. Каким будет вывод следующего кода и почему?

```
>>> def func(a, b, c=3, d=4): print(a, b, c, d)
>>> func(1, *(5, 6))
```

6. А что выведет такой код и почему?

```
>>> def func(a, b, c): a = 2; b[0] = 'x'; c['a'] = 'y'
>>> l=1; m=[1]; n={'a':0}
>>> func(l, m, n)
>>> l, m, n
```

Проверьте свои знания: ответы

- Код выводит 1 2 5, потому что 1 и 2 передаются а и б по позиции, а с опущено в вызове и получает стандартное значение 5.
- Выводом на этот раз будет 1 2 3: значение 1 передается а по позиции, а б и с передаются 2 и 3 по имени (порядок слева направо не играет роли, когда применяются ключевые аргументы, как сделано в коде).
- Код выводит 1 (2, 3), поскольку 1 передается а и *pargs собирает оставшиеся позиционные аргументы в новый объект кортежа. Мы можем проходить по кортежу с добавочными позиционными аргументами посредством любого итерационного инструмента (например, for arg in pargs: ...).
- На этот раз код выводит 1 {'b': 2, 'c': 3}, т.к. 1 передается а по имени и **kargs собирает оставшиеся ключевые аргументы в словарь. Мы можем проходить по ключам словаря с добавочными ключевыми аргументами с помощью любого итерационного инструмента (скажем, for key in kargs: ...). Обратите внимание, что порядок следования ключей в словаре может варьироваться в зависимости от версии Python и других аспектов.
- Выводом здесь будет 1 5 6 4: значение 1 сопоставляется с а по позиции, 5 и 6 сопоставляются с б и с по позиционным аргументам в *имя (6 переопределяет стандартное значение с), а д получает стандартное значение 4, потому что значение для д не передавалось.
- Код выводит (1, ['x'], {'a': 'y'}) – первое присваивание в функции не оказывает влияния на вызывающий код, но остальные два оказывают, поскольку они модифицируют переданный изменяемый объект на месте.

Расширенные возможности функций

В настоящей главе будет представлено собрание более сложных тем, связанных с функциями: рекурсивные функции, атрибуты и аннотации функций, выражение `lambda` и инструменты функционального программирования, такие как `map` и `filter`. Все они являются отчасти расширенными инструментами, с которыми в зависимости от служебных обязанностей вы можете и не встречаться на регулярной основе. Однако из-за своих ролей в определенных предметных областях важно их в целом понимать; например, выражения `lambda` регулярно употребляются в графических пользовательских интерфейсах, а методики функционального программирования получают все большее и большее распространение в коде Python.

Часть искусства применения функций связана с обеспечением их слаженной работы, так что мы исследуем здесь ряд общих принципов проектирования функций. В следующей главе эта более сложная тема будет продолжена рассмотрением генераторных функций и выражений, а также повторением списковых включений в контексте вновь изученных функциональных инструментов.

Концепции проектирования функций

Теперь, когда основы функций в Python уже известны, давайте начнем главу с нескольких слов об имеющейся ситуации. Приступив к серьезному использованию функций, вы столкнетесь с необходимостью выбора того, как увязывать компоненты вместе — скажем, каким образом разложить задачу на содержательные функции (известно как *цепление*), как функции должны взаимодействовать (называется *связностью*) и т.д. Вам также придется учитывать такие понятия, как размер функции, поскольку они напрямую влияют на удобство работы с кодом. Некоторые концепции относятся к категории структурного анализа и проектирования, но они применимы к коду на Python как и на любом другом языке.

Мы ввели ряд идей, касающихся функций и связности модулей, в главе 17 при изучении областей видимости, но ниже приведен обзор нескольких универсальных рекомендаций для читателей, не знакомых с принципами проектирования функций.

- **Связность:** используйте аргументы для входных данных и оператор `return` для выходных данных. Как правило, вы должны стремиться сделать функцию независимой от вещей, находящихся за ее пределами. Аргументы и оператор `return`

часто будут наилучшими способами изоляции внешних зависимостей небольшим количеством хорошо известных мест в коде.

- **Связность:** применяйте глобальные переменные, только когда они по-настоящему нужны. Глобальные переменные (т.е. имена во включающем модуле) обычно являются неудачным способом взаимодействия для функций. Они могут создавать зависимости и проблемы синхронизации, которые затрудняют отладку, изменение и многократное использование программ.
- **Связность:** не модифицируйте изменяемые аргументы, если только такое изменение не ожидается вызывающим кодом. Функции могут модифицировать части передаваемых изменяемых объектов, но это (как и глобальные переменные) создает сильную связность между вызывающим и вызываемым кодом, которая может сделать функцию слишком специфичной и хрупкой.
- **Сцепление:** каждая функция должна иметь единственное унифицированное назначение. При надлежащем проектировании каждая функция должна делать что-то одно — то, что может быть резюмировано в простом повествовательном предложении. Если такое предложение оказывается слишком широким (например, “данная функция реализует всю мою программу”) или содержит много союзов (скажем, “ данная функция дает сотруднику повышение и отправляет заказ пиццы”), тогда имеет смысл подумать о разбиении функции на несколько отдельных более простых функций. В противном случае не удастся повторно применять код, который лежит в основе действий, смешанных в функции.
- **Размер:** каждая функция должна быть относительно небольшой. Цель естественным образом следует из предыдущей цели, но если ваши функции начали занимать несколько страниц на экране редактора, то видимо пришло время их разбить. Прежде всего, учитывая присущую коду Python лаконичность, длинная или глубоко вложенная функция зачастую служит признаком проблем с проектным решением. Сохраняйте функции простыми и короткими.
- **Связность:** избегайте прямого изменения переменных из другого файла модуля. Мы представляли эту концепцию в главе 17 и возвратимся к ней в следующей части книги, когда сосредоточим внимание на модулях. Тем не менее, для справки помните о том, что изменение переменных через границы файлов приводит к появлению связности между модулями подобно тому, как глобальные переменные связывают функции — модули становятся труднее понимать и многократно использовать. При любой возможности применяйте функции доступа вместо прямых операторов присваивания.

На рис. 19.1 иллюстрируются способы взаимодействия функций с внешним миром; входные данные могут поступать из элементов слева, а результаты отправляться в любой форме справа. Опытные проектировщики функций всякий раз, когда возможно, предпочитают использовать для входных данных только аргументы, а для выходных данных операторы `return`.

Конечно, существует масса исключений из перечисленных выше принципов проектирования, включая те, которые имеют отношение к поддержке объектно-ориентированного программирования в Python. Как вы увидите в части VI, классы Python *полагаются* на модификацию переданного изменяемого объекта — функции класса устанавливают атрибуты автоматически передаваемого аргумента по имени `self`, чтобы изменять информацию о состоянии для каждого объекта (например, `self.name='bob'`).

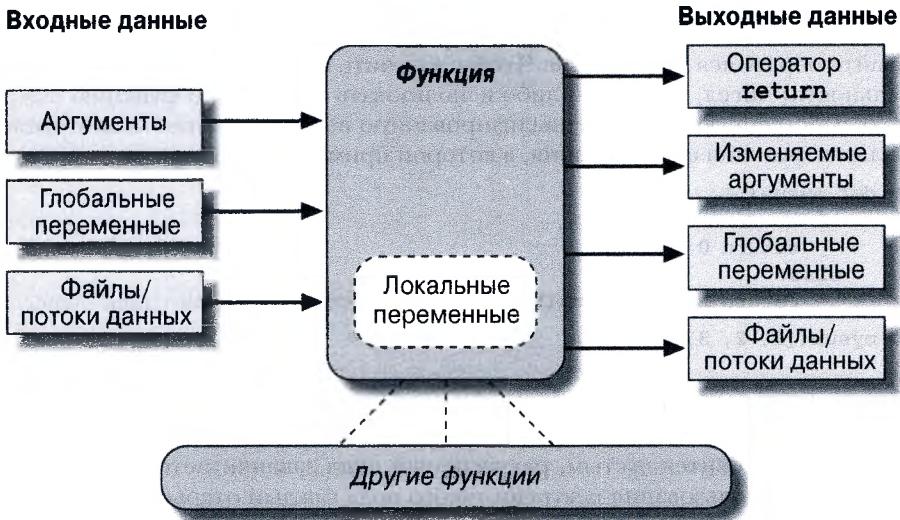


Рис. 19.1. Среда выполнения функций. Функции могут получать входные данные и производить выходные данные разнообразными способами, хотя функции обычно легче восприятии и сопровождении в случае применения аргументов для входных данных и операторов `return` плюс ожидаемой модификации изменяемых аргументов для выходных данных. В Python 3.X выходные данные могут также принимать форму объявленных нелокальных имен, которые существуют в области видимости объемлющих функций

Кроме того, если классы не используются, то глобальные переменные часто оказываются наиболее прямолинейным способом сохранения единственной копии состояния между вызовами для функций в модулях. Побочные эффекты обычно опасны, только если их не ожидают.

Однако в целом вы должны стремиться минимизировать внешние зависимости в функциях и других программных компонентах. Чем более самодостаточной будет функция, тем ее легче понять, многократно применять и модифицировать.

Рекурсивные функции

Мы упоминали о рекурсии при рассмотрении сравнений основных типов в главе 9. Обсуждая правила областей видимости в начале главы 17, мы также кратко отметили, что Python поддерживает *рекурсивные функции* – функции, которые вызывают самих себя либо прямо, либо косвенно с целью организации цикла. В этом разделе мы выясним, на что похожа рекурсия в коде функций.

Рекурсия – довольно сложная тема и ее относительно редко можно встретить в Python, отчасти из-за того, что процедурные операторы Python включают более простые циклические структуры. И все же знать о рекурсии полезно, т.к. она позволяет программам обходить структуры, которые имеют произвольные и непредсказуемые формы и глубины, например, при планировании маршрутов в путешествии, анализе языка и прохождении по ссылкам в веб-сети. Рекурсия даже является альтернативой несложным циклам и итерациям, хотя не обязательно более простой или эффективной.

Суммирование с помощью рекурсии

Давайте обратимся к примерам. Чтобы получить сумму списка (или другой последовательности) чисел, мы можем либо использовать встроенную функцию `sum`, либо написать собственную более специализированную версию. Вот как может выглядеть специальная функция суммирования, в которой применяется рекурсия:

```
>>> def mysum(L):
    if not L:
        return 0
    else:
        return L[0] + mysum(L[1:])    # Рекурсивный вызов самой себя
>>> mysum([1, 2, 3, 4, 5])
15
```

На каждом уровне функция `mysum` рекурсивно вызывает саму себя, чтобы вычислить сумму *остатка* списка, которая позже добавляется к элементу в *голове* списка. Когда список становится пустым, рекурсивный цикл заканчивается и возвращается ноль. В случае использования рекурсии такого рода каждый открытый уровень вызова функции имеет собственную копию локальной области видимости функции в стеке вызовов времени выполнения — здесь это означает, что переменная `L` на каждом уровне разная.

Если код труден для понимания (что часто бывает у новичков), тогда попробуйте добавить в функцию вывод `L` и запустите ее снова, чтобы отследить текущий список на каждом уровне вызова:

```
>>> def mysum(L):
    print(L)      # Трассировка уровней рекурсии
    if not L:    # На каждом уровне L становится короче
        return 0
    else:
        return L[0] + mysum(L[1:])

>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Как легко заметить, суммируемый список на каждом уровне рекурсии становится все меньше, пока окончательно не опустеет — конец рекурсивного цикла. Сумма вычисляется при раскручивании рекурсивных вызовов по возврату.

Альтернативные варианты кода

Интересно, что мы можем сделать код более компактным, применив тернарное выражение `if/else` (описанное в главе 12). Мы также в состоянии обобщить его для любого типа, допускающего суммирование (что легче сделать, если предположить наличие во входных данных, по крайней мере, одного элемента, как делалось в примере с нахождением минимального значения из главы 18), и использовать расширенное присваивание последовательностей Python 3.X, чтобы упростить распаковку первого элемента и остальных (см. главу 11):

```

def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:])          # Использование
                                                       # тернарного выражения

def mysum(L):
    return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # Любой тип,
                                                       # предполагая наличие хотя бы одного элемента

def mysum(L):
    first, *rest = L
    return first if not rest else first + mysum(rest) # Применение расширенного
                                                       # присваивания последовательностей Python 3.X

```

Последние два варианта потерпят неудачу для пустых списков, но допускают последовательности объектов любых типов, поддерживающих операцию +, а не только чисел:

```

>>> mysum([1])           # mysum([]) терпит неудачу в последних двух версиях
1
>>> mysum([1, 2, 3, 4, 5])
15
>>> mysum(['s', 'p', 'a', 'm'])   # Но теперь разрешены разнообразные типы
'spam'
>>> mysum(['spam', 'ham', 'eggs'])
'spamhameggs'

```

Для лучшего понимания запустите все версии самостоятельно. Ниже описано, что выяснится в результате исследования трех вариантов.

- Последние два варианта также работают с одиночным строковым аргументом (например, mysum('spam')), т.к. строки представляют собой последовательности односимвольных строк.
- Третий вариант работает с произвольными итерируемыми объектами, включая открытые входные файлы (mysum(open(имя))), но остальные – нет, потому что они выполняют индексацию (в главе 14 демонстрировалось расширенное присваивание последовательностей на файлах).
- Несмотря на сходство с третьим вариантом, заголовок функции def mysum(first, *rest) вообще не будет работать, поскольку он ожидает передачи отдельных аргументов, а не одиночного итерируемого объекта.

Имейте в виду, что рекурсия может быть прямой, как было показано в примерах до сих пор, или *косвенной*, как в следующем примере (функция, вызывающая другую функцию, которая снова вызывает первую функцию). Совокупный эффект оказывается таким же, хотя на каждом уровне есть два вызова функций вместо одного:

```

>>> def mysum(L):
    if not L: return 0
    return nonempty(L)      # Вызов функции nonempty, которая вызывает mysum

>>> def nonempty(L):
    return L[0] + mysum(L[1:])    # Косвенная рекурсия

>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0

```

Операторы цикла или рекурсия

Несмотря на то что рекурсия работает для суммирования в примерах из предшествующих разделов, в таком контексте она видимо будет излишеством. На самом деле рекурсия в Python не применяется настолько же часто, как в экзотических языках вроде Prolog или Lisp, потому что в Python придается особое значение более простым процедурным операторам наподобие циклов, которые обычно намного естественнее. Скажем, while часто привносит чуть большую конкретику и не требует, чтобы функция определялась как допускающая рекурсивные вызовы:

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> while L:
...     sum += L[0]
...     L = L[1:]
>>> sum
15
```

Еще лучше то, что циклы for обеспечивают автоматическую итерацию, делая рекурсию во многих случаях излишней (и по всей вероятности менее эффективной в плане расхода памяти и времени выполнения):

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> for x in L: sum += x
>>> sum
15
```

В случае использования операторов цикла нам не требуется новая копия локальной области видимости в стеке вызовов для каждой итерации, и мы избегаем затрат времени, связанных с вызовами функций в целом. (В главе 21 будет приведен учебный пример, иллюстрирующий способы измерения времени выполнения для альтернативных реализаций подобного рода.)

Обработка произвольных структур

С другой стороны, рекурсия (или эквивалентные явные алгоритмы, основанные на стеке, которые мы вскоре рассмотрим) может требоваться для обхода структур произвольной формы. В качестве простого примера рекурсии в таком контексте возьмем задачу вычисления суммы всех чисел в структуре с вложенными подсписками следующего вида:

```
[1, [2, [3, 4], 5], 6, [7, 8]]      # Произвольно вложенные под списки
```

Простые операторы циклов здесь не подходят, поскольку это не линейная итерация. Вложенных операторов цикла тоже не будет достаточно, потому что подсписки могут быть вложенными на произвольную глубину и в произвольной форме — нет никакого способа узнать, сколько вложенных циклов необходимо написать для обработки всех случаев. Взамен следующий код приспособливается к такому универсальному вложению за счет применения рекурсии для посещения подсписков:

```
# Файл sumtree.py
def sumtree(L):
    tot = 0
    for x in L:
        if type(x) == list:
            # Для каждого элемента на этом уровне
            tot += sumtree(x)
        else:
            tot += x
    return tot
```

```

if not isinstance(x, list):
    tot += x                                # Сложение чисел напрямую
else:
    tot += sumtree(x)                      # Рекурсия для подсписков
return tot
L = [1, [2, [3, 4], 5], 6, [7, 8]]        # Произвольное вложение
print(sumtree(L)) # Prints 36

# Патологические случаи
print(sumtree([1, [2, [3, [4, [5]]]]])) # Выводит 15 (перегруженное справа)
print(sumtree([[[[[1], 2], 3], 4], 5])) # Выводит 15 (перегруженное слева)

```

Отследите тестовые сценарии в конце кода, чтобы посмотреть, как рекурсия обходит вложенные списки.

Рекурсия или очереди и стеки

Иногда может помочь знание того факта, что внутренне Python реализует рекурсию, заталкивая информацию в стек вызовов при каждом рекурсивном вызове, чтобы запомнить, куда он должен позже возвратиться и продолжить. На самом деле обычно есть возможность реализовать процедуры в рекурсивном стиле без рекурсивных вызовов за счет использования собственного явного стека или очереди для отслеживания оставшихся шагов.

Скажем, приведенный ниже код вычисляет ту же сумму, что и предыдущий пример, но применяет явный список для планирования, когда он посетит элементы в указанном списке, вместо выдачи рекурсивных вызовов; следующим обрабатываемым и суммируемым элементом всегда будет элемент в начале списка:

```

def sumtree(L):                         # Явная очередь с обходом в ширину
    tot = 0
    items = list(L)                      # Начать с копии верхнего уровня
    while items:
        front = items.pop(0)             # Извлечь/удалить элемент в начале
        if not isinstance(front, list):
            tot += front                # Напрямую суммировать числа
        else:
            items.extend(front)          # <== Добавить все из вложенного подсписка
    return tot

```

Формально код обходит список *в ширину* по уровням, потому что он добавляет содержимое вложенных подсписков в конец списка, формируя *очередь* типа “первым пришел – первым обслужен”. Для более точной эмуляции обхода в версии с рекурсивными вызовами мы можем изменить код так, чтобы он выполнял обход *в глубину*, просто добавляя содержимое вложенных подсписков в начало списка и формируя *стек* типа “последним пришел – первым обслужен”:

```

def sumtree(L):                         # Явный стек с обходом в глубину
    tot = 0
    items = list(L)                      # Начать с копии верхнего уровня
    while items:
        front = items.pop(0)             # Извлечь/удалить элемент в начале
        if not isinstance(front, list):
            tot += front                # Напрямую суммировать числа
        else:
            items[:0] = front           # <== Присоединить спереди все из
                                         #   вложенного подсписка
    return tot

```

Код последних двух примеров (и еще одного варианта) находится в файле `sumtree2.py` в пакете примеров для книги. В него добавлено отслеживание элементов списка, чтобы можно было наблюдать за его ростом в обеих схемах, и вывод чисел по мере их посещения, что позволяет видеть порядок поиска. Например, варианты с обходом сначала в ширину и сначала в глубину посещают элементы в тех же трех тестовых списках, которые использовались в рекурсивной версии, в следующих порядках (суммы отображаются последними):

```
c:\code> sumtree2.py
1, 6, 2, 5, 7, 8, 3, 4, 36
1, 2, 3, 4, 5, 15
5, 4, 3, 2, 1, 15
-----
1, 2, 3, 4, 5, 6, 7, 8, 36
1, 2, 3, 4, 5, 15
1, 2, 3, 4, 5, 15
-----
```

Но в целом, как только вы освоитесь с рекурсивными вызовами, они станут более естественными, чем списки с явным планированием, которые они автоматизируют, и обычно более предпочтительными, если только вам не нужно обходить структуру специализированными способами. Скажем, определенные программы выполняют поиск *по первому наилучшему совпадению*, при котором требуется явное дерево поиска, упорядоченное по значимости или по другим критериям. Если вы подумаете о поисковом агенте, который оценивает посещенные веб-страницы по их содержимому, то приложения могут стать гораздо яснее.

Циклы, пути и границы стека

В том виде, как есть, представленных выше программ вполне достаточно в качестве примеров, но крупные рекурсивные приложения временами могут требовать чуть большей инфраструктуры, чем показанная здесь: возможно, понадобится избегать циклов или повторений, записывать пройденные пути для последующего применения и расширять пространство стека, когда используются рекурсивные вызовы вместо явных очередей либо стеков.

Скажем, в рассмотренных примерах с рекурсивными вызовами и явными очередями/стеками ничего не делалось для того, чтобы избежать *циклов* — посещения мест, которые уже были посещены. Здесь это не обязательно, т.к. мы обходили строго иерархические деревья списковых объектов. Тем не менее, если данные могут принимать вид циклического графа, тогда обе схемы потерпят неудачу: версия с рекурсивными вызовами попадет в бесконечный рекурсивный цикл (и привести к исчерпанию пространства в стеке вызовов), а в остальных версиях образуются простые бесконечные циклы, многократно добавляющие те же самые элементы в свои списки (что вполне может вызвать нехватку общей памяти). В некоторых программах необходимо также избегать повторяющейся обработки состояния, достигнутого более одного раза, даже если это не приводит к зацикливанию.

Чтобы добиться успеха, версия с рекурсивными вызовами могла бы просто поддерживать множество, словарь или список состояний, посещенных до сих пор, и в ходе работы проверять его на предмет повторений. Мы будем применять такую схему в примерах с рекурсией позже в книге:

```
if state not in visited:
    visited.add(state)      # x.add(state), x[state]=True или x.append(state)
    ...обработка...
```

В нерекурсивных альтернативах можно было бы аналогичным образом избегать добавления уже посещенных состояний с помощью показанного далее кода. Обратите внимание, что проверка на предмет дубликатов, уже находящихся в списке элементов, позволит избежать планирования какого-то состояния во второй раз, но не предотвратит повторное посещение состояния, которое обходилось ранее и потому удалено из списка:

```
visited.add(front)
...обработка...
items.extend([x for x in front if x not in visited])
```

Представленная модель не совсем применима к сценарию использования в данном разделе, в котором просто суммируются числа в списках, но более крупные приложения будут способны идентифицировать повторяющиеся состояния — например, URL ранее посещенной веб-страницы. Фактически мы будем применять такие методики, чтобы избежать циклов и повторений в примерах, рассматриваемых в следующем разделе.

Некоторые программы могут также нуждаться в записывании полных *путей*, пройденных к каждому состоянию, чтобы по завершении сообщать решения. В подобных случаях каждый элемент в стеке или очереди из нерекурсивной схемы может быть списком полного пути, который достаточен для записи посещенных состояний и содержит следующий элемент, подлежащий исследованию на любом конце.

Также обратите внимание, что стандартный Python ограничивает глубину своего стека вызовов, критически важную для программ с рекурсивными вызовами, чтобы отлавливать ошибки бесконечной рекурсии. Для расширения стека используйте модуль `sys`:

```
>>> sys.getrecursionlimit() # По умолчанию глубина составляет 1000 вызовов
1000
>>> sys.setrecursionlimit(10000) # Делает возможным более глубокое вложение
>>> help(sys.setrecursionlimit) # Дополнительные сведения об этой настройке
```

Максимально допустимое значение для настройки может варьироваться в зависимости от платформы. Она не требуется для программ, в которых применяются стеки или очереди, чтобы избежать рекурсивных вызовов и получить больший контроль над процессом обхода.

Дополнительные примеры рекурсии

Несмотря на искусственность примера в данном разделе, он является типичным представителем более широкого класса программ; скажем, деревья наследования классов и цепочки импортирования модулей могут показывать похожие общие структуры, а вычислительные структуры вроде перестановок могут требовать произвольно много вложенных циклов. Позже в книге мы снова будем использовать рекурсию в более реалистичных примерах:

- в файле `permute.py` из главы 20 для тасования произвольных последовательностей;
- в файле `reloadall.py` из главы 25 для обхода цепочек импортирования модулей;
- в файле `classtree.py` из главы 29 для обхода деревьев наследования классов;
- в файле `lister.py` из главы 31 опять для обхода деревьев наследования классов;
- в решениях двух упражнений для этой части книги (обратного отсчета и вычисления факториала), приведенных в приложении.

Во втором и третьем примере будут также обнаруживаться уже посещенные состояния, чтобы избежать циклов и повторений. Хотя для линейных итераций в целом следует отдавать предпочтение обычным циклам по причине их простоты и эффективности, мы обнаружим, что рекурсия будет неотъемлемой в сценариях, подобных упомянутым выше примерам.

Более того, иногда необходимо знать о возможности *непреднамеренной* рекурсии в своих программах. Как будет показано позже в книге, некоторые методы перегрузки операций в классах, такие как `__setattr__` и `__getattribute__` и даже `__repr__`, обладают потенциалом рекурсивного зацикливания при некорректном применении. Рекурсия — мощный инструмент, но она проявляет себя наилучшим образом, когда ее понимают и ожидают!

Объекты функций: атрибуты и аннотации

Функции Python гораздо гибче, чем может казаться. Как выяснилось в этой части книги, функции в Python представляют собой намного большее, чем спецификации по генерации кода для компилятора — функции Python являются полноценными *объектами*, хранящимися в собственных участках памяти. Будучи таковыми они могут свободно передаваться внутри программы и вызываться косвенно. Они также поддерживают операции, которые вообще не имеют отношения к вызовам, а связаны с хранилищем атрибутов и аннотаций.

Косвенные вызовы функций: "первоклассные" объекты

Из-за того, что функции Python представляют собой объекты, вы можете писать программы, которые обрабатывают их обобщенным образом. Объекты функций можно присваивать другим именам, передавать другим функциям, встраивать в структуры данных, возвращать из одной функции в другой и т.п., как если бы они были простыми числами или строками. Кроме того, объекты функций поддерживают специальную операцию: их можно вызывать, перечисляя аргументы в круглых скобках после выражения функции. И все-таки функции относятся к той же самой общей категории, что и остальные объекты.

Обычно это называют *первоклассной объектной моделью*; в Python она вездесуща и является необходимой частью функционального программирования. Мы исследуем такой режим программирования более полно в текущей и следующей главах; из-за того, что его основная идея зиждется на понятии применения функций, функции должны трактоваться как данные.

Мы уже видели некоторые из обобщенных сценариев использования для функций в более ранних примерах, но краткий обзор поможет акцентировать внимание на объектной модели. Скажем, с указываемым в операторе `def` именем не связано ничего необычного: оно представляет собой всего лишь переменную, присваиваемую в текущей области видимости, как если бы она находилась слева от знака `=`. После выполнения оператора `def` имя функции становится просто ссылкой на объект — вы можете свободно присваивать данный объект другим именам и вызывать его через любую ссылку:

```
>>> def echo(message):      # Имя echo присваивается объект функции
    print(message)

>>> echo('Direct call')   # Вызов объекта через первоначальное имя
Direct call
```

```
>>> x = echo          # Теперь x тоже ссылается на объект функции
>>> x('Indirect call!')  # Вызов объекта через имя x путем добавления ()
Indirect call!
```

Поскольку аргументы передаются по присваиванию объектов, функции легко передавать другим функциям в качестве аргументов. Вызываемая функция затем может вызвать переданную функцию, просто добавив аргументы в круглых скобках:

```
>>> def indirect(func, arg):
    func(arg)           # Вызов переданного объекта путем добавления ()

>>> indirect(echo, 'Argument call!')  # Передача функции другой функции
Argument call!
```

Объекты функций можно даже помещать внутрь структур данных, как если бы они были целыми числами или строками. Например, в следующем коде функция дважды встраивается в список кортежей, который исполняет роль таблицы действий. Из-за того, что составные типы Python подобного рода способны содержать объекты любых видов, здесь также нет какого-то особых случая:

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
    func(arg)      # Вызов функций, встроенных в контейнер
Spam!
Ham!
```

В коде просто производится проход по списку `schedule` и вызов функции `echo` с одним аргументом на каждой итерации (обратите внимание в заголовке цикла `for` на распаковывающее присваивание кортежа, представленное в главе 13). Как демонстрировали примеры в главе 17, функции также можно создавать и возвращать для применения где-то в другом месте — *замыкания*, созданные в таком режиме, еще и предохраняют состояние из объемлющей области видимости:

```
>>> def make(label):    # Создание функции без ее вызова
    def echo(message):
        print(label + ':' + message)
    return echo

>>> F = make('Spam')    # label в объемлющей области видимости предохраняется
>>> F('Ham!')          # Вызов функции, возвращенной make
Spam:Ham!
>>> F('Eggs!')
Spam:Eggs!
```

Универсальная первоклассная объектная модель и отсутствие объявлений типов делают Python невероятно гибким языком программирования.

Интроспекция функций

Поскольку функции являются объектами, мы можем их обрабатывать посредством нормальных инструментов для объектов. На самом деле функции оказываются более гибкими, чем можно было ожидать. Скажем, после создания функцию можно вызывать как обычно:

```
>>> def func(a):
    b = 'spam'
    return b * a

>>> func(8)
'spamspamspamspamspamspamspam'
```

Но выражение вызова — только одна операция, определенная для работы на объектах функций. Мы можем также инспектировать их атрибуты обобщенным образом (следующий код запускался в Python 3.7, но результаты в Python 2.X похожи):

```
>>> func.__name__
'func'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...остальные не показаны: всего их 35...
['__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

Инструменты для интроспекции позволяют нам исследовать также детали реализации — например, функции имеют присоединенные *кодовые объекты*, которые представляют подробные сведения о таких аспектах, как локальные переменные и аргументы функций:

```
>>> func.__code__
<code object func at 0x00000000021A6030, file "<stdin>", line 1>
>>> dir(func.__code__)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
...остальные не показаны: всего их 38...
['co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']

>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
1
```

Разработчики инструментов могут задействовать информацию такого рода для управления функциями (фактически мы сделаем это в главе 39, чтобы реализовать проверку достоверности аргументов функций в декораторах).

Атрибуты функций

Объекты функций не ограничены системными атрибутами, перечисленными в предыдущем разделе. Как было указано в главе 17, начиная с версии Python 2.1, к ним можно присоединять произвольные атрибуты, *определяемые пользователем*:

```
>>> func
<function func at 0x000000000296A1E0>
>>> func.count = 0
>>> func.count += 1
>>> func.count
1
>>> func.handles = 'Button-Press'
>>> func.handles
'Button-Press'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...и т.д.: в Python 3.X все остальные имена имеют двойные подчеркивания,
так что ваши имена с ними не конфликтуют...
['__str__', '__subclasshook__', 'count', 'handles']
```

Собственные связанные с реализацией данные Python, хранящиеся с объектами функций, следуют соглашениям об именовании, которые предотвращают конфликты с более произвольными именами атрибутов, которые вы можете назначать самостоятельно. В Python 3.X все имена внутренностей функций содержат ведущие и хвосто-

вые двойные подчеркивания (`__X__`); в Python 2.X соблюдается та же схема, но некоторые имена начинаются с `func_X`:

```
c:\code> py -3
>>> def f(): pass
>>> dir(f)
...запустите самостоятельно, чтобы просмотреть...
>>> len(dir(f))
35
>>> [x for x in dir(f) if not x.startswith('__')]
[]

c:\code> py -2
>>> def f(): pass
>>> dir(f)
...запустите самостоятельно, чтобы просмотреть...
>>> len(dir(f))
31
>>> [x for x in dir(f) if not x.startswith('__')]
['func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc',
'func_globals', 'func_name']
```

Если вы стараетесь не именовать атрибуты одинаково, тогда можете благополучно использовать пространство имен функции, как если бы оно было собственным пространством имен или областью видимости.

В главе было показано, что такие атрибуты могут применяться для присоединения информации о состоянии непосредственно к объектам функций вместо использования других методик, подобных глобальным переменным, нелокальным переменным и классам. В отличие от нелокальных переменных атрибуты доступны везде, где доступна сама функция, даже извне ее кода.

В определенном смысле это также способ эмуляции “статических локальных переменных” в других языках — переменных, имена которых локальны для функции, но значения предохраняются после окончания работы функции. Атрибуты относятся к объектам, а не к областям видимости (и на них нужно ссылаться через имя функции внутри ее кода), но совокупный эффект похож.

Более того, как было показано в главе 17, когда атрибуты присоединяются к функциям, генерируемым другими *фабричными* функциями, они также поддерживают сохранение состояния с возможностью записи и множеством копий для каждого вызова во многом подобно нелокальным замыканиям и атрибутам экземпляров классов.

Аннотации функций в Python 3.X

В Python 3.X (но не в Python 2.X) также допускается присоединять к объекту функции *аннотирующую информацию* — произвольные определяемые пользователем данные об аргументах и результате функции. Для указания аннотаций Python предлагает специальный синтаксис, но он ничего не делает с самими аннотациями; они совершенно необязательны и когда присутствуют, то просто присоединяются к атрибуту `__annotations__` объекта функции для применения другими инструментами. Например, такие инструменты могут использовать аннотации в контексте проверки на предмет ошибок.

В предыдущей главе мы узнали об аргументах с передачей только по ключевым словам в Python 3.X; аннотации дополнительно обобщают синтаксис заголовков фун-

кций. Взгляните на следующую неаннотированную функцию, которая принимает три аргумента и возвращает результат:

```
>>> def func(a, b, c):
    return a + b + c
>>> func(1, 2, 3)
6
```

Синтаксически аннотации функций записываются в строках заголовков `def` как произвольные выражения, ассоциированные с аргументами и возвращаемыми значениями. Для аргументов они указываются после двоеточия, следующего непосредственно за именем аргумента; для возвращаемых значений они помещаются после символов `->`, следующих за списком аргументов. Скажем, в приведенном далее коде аннотируются все три аргумента предыдущей функции, а также ее возвращаемое значение:

```
>>> def func(a: 'spam', b: (1, 10), c: float) -> int:
    return a + b + c
>>> func(1, 2, 3)
6
```

Вызовы аннотированной функции работают обычным образом, но при наличии аннотации Python накапливает их в *словаре* и присоединяет его к самому объекту функции. Имена аргументов становятся ключами, аннотация возвращаемого значения, если имеется, то сохраняется под ключом `return` (чего достаточно, т.к. ключевое слово `return` не может применяться для имени аргумента), и значения ключей аннотации присваиваются результатам выражений аннотаций:

```
>>> func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'spam', 'return': <class 'int'>}
```

Поскольку аннотации являются всего лишь объектами Python, присоединенными к объекту Python, обрабатывать их просто. В следующем коде аннотируются только два из трех аргументов, а также производится проход по присоединенным аннотациям обобщенным образом:

```
>>> def func(a: 'spam', b, c: 99):
    return a + b + c
>>> func(1, 2, 3)
6
>>> func.__annotations__
{'c': 99, 'a': 'spam'}
>>> for arg in func.__annotations__:
    print(arg, '=>', func.__annotations__[arg])
c => 99
a => spam
```

Здесь необходимо отметить две тонкости. Во-первых, вы по-прежнему можете использовать *стандартные значения* для аргументов в случае снабжения их аннотациями – аннотация (и ее символ `:`) появляется перед стандартным значением (и его символом `=`). В показанном ниже коде `a: 'spam' = 4` означает, что аргумент `a` имеет стандартное значение 4 и аннотирован строкой `'spam'`:

```
>>> def func(a: 'spam' = 4, b: (1, 10) = 5, c: float = 6) -> int:
    return a + b + c
>>> func(1, 2, 3)
6
```

```
>>> func()          # 4 + 5 + 6 (все стандартные значения)
15
>>> func(1, c=10)   # 1 + 5 + 10 (ключевые аргументы нормально работают)
16
>>> func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'spam', 'return': <class 'int'>}
```

Во-вторых, обратите внимание на то, что все *пробелы* в предыдущем примере не обязательны — вы можете применять или не применять пробелы между компонентами в заголовках функций, но их отсутствие может ухудшить читабельность кода для одних (и возможно улучшить ее для других!):

```
>>> def func(a:'spam'=4, b:(1,10)=5, c:float=6)->int:
    return a + b + c
>>> func(1, 2)      # 1 + 2 + 6
9
>>> func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'spam', 'return': <class 'int'>}
```

Аннотации являются новым средством в Python 3.X, и ряд их потенциальных сценариев использования еще предстоит раскрыть. Однако легко представить себе применение аннотаций с целью указания ограничений для типов и значений аргументов, а более крупные API-интерфейсы могли бы использовать это средство как способ регистрации информации об интерфейсах функций.

Фактически в главе 39 мы рассмотрим потенциальное приложение, в котором аннотации послужат альтернативой *аргументам декораторов функций* — более общей идеи, предусматривающей запись информации вне заголовка функции, так что она не ограничивается единственной ролью. Подобно самому Python аннотации являются инструментом, роли которого формируются лишь вашим воображением.

Наконец, имейте в виду, что аннотации работают только в операторах `def`, но не в выражениях `lambda`, т.к. синтаксис `lambda` уже ограничивает полезность определяемых с его помощью функций. Итак, мы подходим к нашей следующей теме.

Анонимные функции: выражения `lambda`

Помимо оператора `def` в Python также предлагается форма выражения, генерирующая объекты функций. Из-за сходства с инструментом в языке Lisp она называется `lambda`¹. Подобно `def` такое выражение создает функцию, которая будет вызываться позже, но возвращает сам объект функции, не присваивая его имени. По этой причине выражения `lambda` иногда называют *анонимными* (т.е. безымянными) функциями. На практике они часто применяются для того, чтобы встроить определение функции в строку или отложить выполнение порции кода.

¹ Понятие `lambda` отчего-то пугает людей больше, чем должно. Похоже, такая реакция проистекает из самого названия “`lambda`” (лямбда) — имени, пришедшего из языка Lisp, который получил его от лямбда-исчисления, представляющего собой форму символической логики. Тем не менее, в Python на самом деле это всего лишь ключевое слово, которое синтаксически вводит выражение. Оставив в стороне математическое наследие, выражение `lambda` проще в использовании, нежели может показаться: оно просто является альтернативным способом написания кода функции, хотя и без полных операторов, декораторов или аннотаций Python 3.X.

Основы выражения lambda

Общая форма `lambda` выглядит как ключевое слово `lambda`, за которым следует один или больше аргументов (очень похоже на список аргументов, заключенный в круглые скобки в заголовке `def`) и далее выражение после двоеточия:

`lambda аргумент1, аргумент2, ... аргументN : выражение, использующее аргументы`

Объекты функций, возвращаемые выражениями `lambda`, работают в точности как объекты функций, создаваемые и присваиваемые операторами `def`, но существует несколько отличий, которые делают выражения `lambda` полезными в специализированных ролях.

- `lambda` представляет собой выражение, а не оператор. По этой причине выражение `lambda` может находиться в местах, где оператор `def` не разрешен синтаксисом Python — скажем, внутри спискового литерала или в аргументах вызова функции. В случае `def` на функции можно ссылаться по именам, но они должны быть созданы где-то в другом месте. Как выражение, `lambda` возвращает значение (новую функцию), которую дополнительно можно присвоить имени. Напротив, оператор `def` всегда присваивает новую функцию имени в своем заголовке вместо возвращения ее в виде результата.
- Тело `lambda` является одиночным выражением, а не блоком операторов. Тело `lambda` похоже на то, что было бы указано в операторе `return` внутри тела `def`; вы просто набираете результат как обычное выражение, не возвращая его явно. Из-за ограничения только выражением тело `lambda` менее универсально, чем `def` — вы можете помещать в тело `lambda` лишь определенную логику, не использующую операторы `while` и `if`. Так было задумано, чтобы ограничить вложенность в программе: выражение `lambda` предназначено для записи простых функций, а оператор `def` поддерживает более крупные задачи.

За исключением описанных отличий операторы `def` и выражения `lambda` выполняют тот же самый вид работы. Например, ранее было показано, как создать функцию посредством оператора `def`:

```
>>> def func(x, y, z): return x + y + z
>>> func(2, 3, 4)
9
```

Но достичь того же эффекта можно с помощью выражения `lambda`, явно присваивая его результат имени, через которое позже будет вызываться функция:

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

Здесь имени `f` присваивается объект функции, созданный выражением `lambda`; так работает оператор `def`, но он производит присваивание автоматически.

Как и в случае `def`, для аргументов `lambda` можно указывать стандартные значения:

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefloe'
```

Код в теле `lambda` также следует тем же самим правилам поиска в областях видимости, что и код внутри `def`. Выражения `lambda` вводят локальную область видимости очень похоже на вложенный оператор `def`, которая автоматически видит имена

из областей видимости объемлющих функций, модуля и встроенной (через правило LEGB, обсуждавшееся в главе 17):

```
>>> def knights():
    title = 'Sir'
    action = (lambda x: title + ' ' + x)      # title из области видимости
                                                # объемлющего def
    return action                                # Возвращение объекта функции

>>> act = knights()
>>> msg = act('robin')                         # 'robin' передается аргументу x
>>> msg
'Sir robin'

>>> act                                         # act: функция, а не ее результат
<function knights.<locals>.lambda at 0x00000000029CA488>
```

До выхода версии Python 2.2 в приведенном примере значение для имени `title` обычно взамен передавалось как стандартное значение аргумента; если вы подзабыли области видимости, тогда перечитайте главу 17.

Для чего используется выражение `lambda`?

Вообще говоря, выражение `lambda` полезно как своего рода краткое условное обозначение функции, которое позволяет встраивать определение функции внутрь кода, где оно применяется. Выражение `lambda` совершенно необязательно – вы всегда можете вместо него использовать оператор `def` и должны поступать так, если функция требует мощи полных операторов, которую выражение `lambda` не способно легко предоставить. Но в сценариях, где нужно всего лишь встраивать небольшие порции исполняемого кода в местах их применения, выражения `lambda` окажутся более простыми кодовыми конструкциями.

Скажем, позже мы выясним, что обработчики обратных вызовов часто записываются как внутристоронние выражения `lambda`, встроенные прямо в список аргументов регистрирующего вызова, взамен их определения с помощью `def` где-то в коде и ссылки по имени (см. врезку “Что потребует внимания: обратные вызовы `lambda`” далее в главе).

Выражения `lambda` также широко используются при реализации *таблиц переходов*, которые представляют собой списки или словари действий, подлежащих выполнению по запросу. Вот пример:

```
L = [lambda x: x ** 2,          # Внутристороннее определение функции
      lambda x: x ** 3,
      lambda x: x ** 4]           # Список из трех вызываемых функций

for f in L:
    print(f(2))                # Выводит 4, 8, 16
print(L[0](3))                 # Выводит 9
```

Выражение `lambda` наиболее полезно в качестве сокращения для `def`, когда необходимо размещать небольшие порции исполняемого кода там, где операторы запрещены синтаксисом. Например, в предыдущем фрагменте кода строится список из трех функций за счет встраивания выражений `lambda` в списоковый литерал; внутри списковых литералов такого рода `def` не допускается, поскольку он является оператором, а не выражением. Эквивалентный код с `def` потребовал бы временных имен функций (они могли бы конфликтовать с остальными именами) и определений функций за пределами контекста их планируемого применения (который может находиться на сотни строк дальше):

```

def f1(x): return x ** 2
def f2(x): return x ** 3
def f3(x): return x ** 4
# Определение именованных функций

L = [f1, f2, f3]           # Ссылка по имени
for f in L:
    print(f(2))            # ВЫВОДИТ 4, 8, 16
print(L[0](3))             # ВЫВОДИТ 9

```

Переключатели для множественного ветвления: окончание

На самом деле вы можете использовать словари и другие структуры данных в Python для построения более универсальных разновидностей таблиц действий. Вот еще один пример, выполненный в интерактивной подсказке:

```

>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
     'got': (lambda: 2 * 4),
     'one': (lambda: 2 ** 6)}[key]()
8

```

Когда Python создает временный словарь, каждое вложенное выражение `lambda` генерирует и оставляет после себя функцию, подлежащую вызову в будущем. Индексация по ключу извлекает одну из функций, а круглые скобки приводят к вызову извлеченной функции. При такой реализации словарь становится более универсальным инструментом для множественного ветвления, нежели показанный в главе 12 во время обсуждения операторов `if`.

Чтобы написать версию кода без `lambda`, понадобится предусмотреть три оператора `def` где-то в файле за пределами словаря, в котором функции будут применяться, и ссылаться на функции по их именам:

```

>>> def f1(): return 2 + 2
>>> def f2(): return 2 * 4
>>> def f3(): return 2 ** 6
>>> key = 'one'
>>> {'already': f1, 'got': f2, 'one': f3}[key]()
64

```

Такой код тоже работает, но операторы `def` могут находиться произвольно далеко в файле, даже если они представляют собой лишь небольшие порции кода. *Кодовая близость*, обеспечиваемая выражениями `lambda`, особенно полезна для функций, которые будут использоваться только в одном контексте — если эти три функции больше нигде не нужны, то имеет смысл встроить их определения внутрь словаря как выражения `lambda`. Кроме того, форма `def` требует выбора для функций имен, которые могут конфликтовать с остальными именами в файле (маловероятно, но всегда возможно)².

² Однажды один студент заметил, что от словаря с координирующей таблицей в таком коде можно было бы избавиться, если бы имя функции совпадало со своим строковым ключом поиска — запуск `eval('имя_функции')()` инициирует вызов. Хотя в данном случае это справедливо и временами удобно, как было показано ранее (скажем, в главе 10), функция `eval` работает относительно медленно (она обязана скомпилировать и выполнить код), к тому же небезопасна (вы должны доверять источнику строки). По существу таблицы переходов обычно задействуются диспетчеризацией полиморфных методов в Python: вызов метода делает “то, что надо”, основываясь на типе объекта. В части VI будет объясняться, почему.

Выражения `lambda` также оказываются полезными в списках аргументов с вызовами функций как способ встраивания определений временных функций, которые не применяются в каких-то других местах программы; примеры такого использования будут рассматриваться при исследовании `map` позже в главе.

Как (не) запутать свой код на Python

Тот факт, что тело `lambda` обязано быть одиночным выражением (не серией операторов), казалось бы, накладывает строгие ограничения на объем логики, которую можно упаковывать в `lambda`. Однако если вы знаете, что делаете, тогда можете записывать большинство операторов Python в виде эквивалентов, основанных на выражениях.

Например, если вы хотите выводить из функции `lambda`, то просто напишите код `print(X)` в Python 3.X, где он становится выражением вызова, а не оператором, или `sys.stdout.write(str(X) + '\n')` в Python 2.X или 3.X, чтобы обеспечить переносимость выражения (вспомните из главы 11, что именно это и делает `print`). Подобным же образом для вложения в `lambda` логики *выбора* вы можете применять тернарное выражение `if/else`, описанное в главе 12, либо эквивалентную, но более сложную комбинацию `and/or`, которая также обсуждалась в главе 12. Как вам уже известно, следующий оператор:

```
if a:  
    b  
else:  
    c
```

можно эмулировать посредством одного из двух почти эквивалентных выражений:

```
b if a else c  
((a and b) or c)
```

Поскольку выражения такого рода можно помещать внутрь `lambda`, их можно использовать для реализации логики выбора в функции `lambda`:

```
>>> lower = (lambda x, y: x if x < y else y)  
>>> lower('bb', 'aa')  
'aa'  
>>> lower('aa', 'bb')  
'aa'
```

Более того, если необходимо организовать *циклы* внутри `lambda`, то можно внедрять вызовы `map` и выражения списковых включений — инструменты, с которыми мы встречались в предшествующих главах и которые более подробно обсудим в следующей главе:

```
>>> import sys  
>>> showall = lambda x: list(map(sys.stdout.write, x))      # Python 3.X:  
                                         # должен использоваться список  
>>> t = showall(['spam\n', 'toast\n', 'eggs\n'])           # Python 3.X:  
                                         # может применяться print  
spam  
toast  
eggs  
>>> showall = lambda x: [sys.stdout.write(line) for line in x]  
>>> t = showall(['bright\n', 'side\n', 'of\n', 'life\n'])  
bright  
side
```

```

of
life
>>> showall = lambda x: [print(line, end='') for line in x] # То же самое:
                                         # только Python 3.X
>>> showall = lambda x: print(*x, sep='', end='')      # То же самое:
                                         # только Python 3.X

```

Существует ограничение на эмуляцию операторов с помощью выражений: скажем, вы не можете напрямую достичь эффекта оператора присваивания, хотя иногда подходит инструменты вроде встроенной функции `setattr`, атрибута `__dict__` с привязками пространств имен и методов, которые модифицируют изменяемые объекты на месте, а методики функционального программирования способны увести вас в темное королевство мудреных выражений.

Теперь, продемонстрировав такие трюки, я обязан попросить вас применять их только в крайнем случае. Без должного старания они могут привести к нечитабельному (т.е. запутанному) коду на Python. В общем и целом простое лучше сложного, явное лучше неявного и полные операторы лучше загадочных выражений. Вот почему `lambda` ограничивается выражениями. При необходимости написания более крупной логики используйте оператор `def`; выражение `lambda` предназначено для небольших порций внутристорочного кода. С другой стороны, при умеренном применении вы можете найти эти методики удобными.

Области видимости: выражения `lambda` также могут быть вложенными

Выражения `lambda` являются главными выгодоприобретателями поиска в областях видимости вложенных функций (*E* в правиле LEGB из главы 17). В качестве проверки далее выражение `lambda` находится внутри `def` – типичный случай – и потому способно получать доступ к значению, которое имя `x` имело в области видимости объемлющей функции на момент, когда эта функция вызывалась:

```

>>> def action(x):
    return (lambda y: x + y) # Создание и возвращение функции, запоминание x
>>> act = action(99)
>>> act
<function action.<locals>.<lambda> at 0x000000000029CA2F0>
>>> act(2)                # Вызов тогс, что возвратила action
101

```

В предыдущем обсуждении областей видимости вложенных функций не был проиллюстрирован тот факт, что функция `lambda` также имеет доступ к именам в любой объемлющей функции `lambda`. Данный случай не вполне ясен, но вообразите, что мы перепишем предшествующий оператор `def` с использованием `lambda`:

```

>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
>>> act(3)
102
>>> ((lambda x: (lambda y: x + y))(99))(4)
103

```

Здесь вложенная структура `lambda` создает функцию, которая при вызове также создает функцию. В обеих ситуациях код вложенной конструкции `lambda` имеет доступ к переменной `x` из объемлющей конструкции `lambda`. Хотя код работает, он выглядит довольно запутанным; в интересах читабельности вложенных `lambda` в целом лучше избегать.

Что потребует внимания: обратные вызовы `lambda`

Еще одно очень распространенное приложение `lambda` связано с определением внутристоронних функций обратного вызова при работе с модулем `tkinter`, предназначенный для создания графических пользовательских интерфейсов в Python (в Python 2.X он называется Tkinter). Скажем, в следующем коде создается кнопка, которая по щелчку выводит на консоль сообщение, исходя из предположения, что модуль `tkinter` доступен на компьютере (по умолчанию это так в средах Windows, Mac, Linux и других операционных систем):

```
import sys
from tkinter import Button, mainloop # Tkinter в Python 2.X
x = Button(
    text='Press me',
    command=(lambda:sys.stdout.write('Spam\n'))) # Python 3.X: print()
x.pack() # В консольном режиме может быть необязательным
mainloop()
```

В коде регистрируется обработчик обратного вызова путем передачи функции, сгенерированной с помощью `lambda`, в ключевом аргументе `command`. Преимущество `lambda` перед `def` в данной ситуации связано с тем, что код, который обрабатывает щелчок на кнопке, находится прямо здесь в вызове, создающем кнопку.

В действительности `lambda` откладывает выполнение обработчика до момента возникновения события: вызов `write` происходит при щелчке на кнопке, а не когда кнопка создана, и фактически “знает” строку, которую он должен записать при появлении события.

Поскольку правила поиска в областях видимости вложенных функций применимы также к выражениям `lambda`, начиная с версии Python 2.2, их легче использовать в качестве обработчиков обратных вызовов — они автоматически видят имена из функций, в которых написаны, и в большинстве случаев уже не требуют передачи стандартных значений. Это особенно удобно для доступа к специальному аргументу экземпляра `self`, который является локальной переменной в функциях методов объемлющего класса (классы рассматриваются в части VI):

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.onPress("spam")))
    def onPress(self, message):
        ...использование message...
```

В ранних версиях Python даже `self` приходилось передавать в `lambda` посредством стандартных значений. Как будет показано позже, объекты классов с методами `__call__` и `__new__` также часто исполняют роли обратных вызовов (см. главы 30 и 31).

Инструменты функционального программирования

Согласно большинству определений современный Python сочетает в себе поддержку множества парадигм программирования: процедурной (посредством своих базовых операторов), объектно-ориентированной (с помощью классов) и функциональной. Для последней парадигмы Python включает набор встроенных функций, используемых при *функциональном программировании* — инструментов, которые применяют функции к последовательностям и другим итерируемым объектам. В состав набора

входят инструменты, которые вызывают функции на элементах итерируемого объекта (`map`), фильтруют элементы на основе проверяемой функции (`filter`) и применяют функции к парам элементов и результатов выполнения (`reduce`).

Несмотря на то что границы временами слегка размыты, в соответствие с большинством определений арсенал функционального программирования Python также содержит исследованную выше *первоначальную объектную модель*, обсуждавшиеся ранее замыкания вложенных областей видимости и анонимные функции `lambda`, *генераторы* и *включения*, которые подробно рассматриваются в следующей главе, и возможно *декораторы* функций и классов из финальной части книги. В заключение главы мы приведем краткий обзор встроенных функций, автоматически применяющих другие функции к итерируемым объектам.

Отображение функций на итерируемые объекты: `map`

Одним из наиболее часто встречающихся действий, выполняемых в программах со списками и другими последовательностями, является применение какой-то операции к каждому элементу и накопление результатов — выбор столбцов в таблицах базы данных, увеличение значений в полях с заработной платой сотрудников компании, разбор вложений в сообщениях электронной почты и т.д. В Python есть много инструментов, которые облегчают написание кода таких операций в масштабах целой коллекции. Например, обновление всех счетчиков в списке несложно обеспечить в цикле `for`:

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)      # Добавить 10 к каждому элементу
>>> updated
[11, 12, 13, 14]
```

Но из-за того, что эта операция употребляется настолько широко, Python также предоставляет встроенную функцию, которая выполняет большую часть работы. Встроенная функция `map` применяет переданную функцию к каждому элементу в итерируемом объекте и возвращает список, содержащий все результаты вызовов функции. Вот пример:

```
>>> def inc(x): return x + 10      # Функция, подлежащая выполнению
>>> list(map(inc, counters))       # Накапливание результатов
[11, 12, 13, 14]
```

В главах 13 и 14 мы кратко упоминали о `map` как о способе применения какой-нибудь встроенной функции к элементам в итерируемом объекте. Здесь мы обобщим ее использование за счет передачи функции, определяемой пользователем, которая подлежит применению к каждому элементу в списке — `map` вызывает `inc` на каждом элементе списка и собирает все возвращаемые значения в новый список. Вспомните, что `map` в Python 3.X является итерируемым объектом, поэтому для вывода всех результатов используется вызов `list`, который в Python 2.X необязателен (см. главу 14, если вы забыли о таком требовании).

Поскольку встроенная функция `map` ожидает передачи функции, применяемой к элементам, она также относится к тем местам, где обычно появляется выражение `lambda`:

```
>>> list(map(lambda x: x + 3, counters))    # Выражение функции
[4, 5, 6, 7]
```

Подлежащая применению функция добавляет 3 к каждому элементу в списке counters; так как эта небольшая функция больше нигде не нужна, она была записана как внутристрочная посредством выражения `lambda`. Из-за того, что такие использования `map` эквиваленты цикла `for`, добавив немного кода, вы можете получить универсальную утилиту отображения:

```
>>> def mymap(func, seq):
    res = []
    for x in seq: res.append(func(x))
    return res
```

Предполагая, что функция `inc` осталась в прежнем виде, мы можем отобразить ее на последовательность (или другой итерируемый объект) с помощью встроенной функции или нашего эквивалента:

```
>>> list(map(inc, [1, 2, 3])) # Встроенная функция является итерируемым объектом
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])    # Наша функция строит список (см. генераторы)
[11, 12, 13]
```

Тем не менее, так как `map` – встроенная функция, она всегда доступна, всегда работает одинаково и дает преимущества в плане производительности (как будет доказано в главе 21, в некоторых режимах применения она быстрее вручную написанного цикла `for`). Кроме того, функцию `map` можно использовать более развитыми способами, чем показано здесь. Скажем, получив в качестве аргументов несколько последовательностей, `map` передает извлеченные из последовательностей элементы как индивидуальные аргументы функции `pow`:

```
>>> pow(3, 4)                                # 3**4
81
>>> list(map(pow, [1, 2, 3], [2, 3, 4]))    # 1**2, 2**3, 3**4
[1, 8, 81]
```

В случае множества последовательностей `map` ожидает функцию с N аргументами для N последовательностей. Функция `pow` принимает в каждом вызове два аргумента – по одному из каждой последовательности, переданной `map`. Эмуляция такого обобщения на множество последовательностей в коде требует не особенно много работы, но мы отложим ее выполнение до следующей главы, где будут представлены дополнительные инструменты итерации.

Вызов `map` похож на выражения списковых включений, которые мы исследовали в главе 14 и в следующей главе снова к ним обратимся с точки зрения функционального программирования:

```
>>> list(map(inc, [1, 2, 3, 4]))
[11, 12, 13, 14]
>>> [inc(x) for x in [1, 2, 3, 4]] # Для генерации элементов взамен используйте ()
[11, 12, 13, 14]
```

В ряде случаев `map` способна выполнять быстрее, чем списковое включение (например, когда отображается встроенная функция), и может также требовать меньшего объема кода. С другой стороны, поскольку `map` применяет к каждому элементу вызов `функции`, а не произвольное `выражение`, она является менее универсальным инструментом и часто требует добавочных вспомогательных функций или `lambda`. Вдобавок помещение включения в круглые скобки вместо квадратных приводит к созданию объекта, который `генерирует` значения по запросу с целью экономии памяти и увеличения скорости реагирования, что во многом похоже на `map` в Python 3.X – мы займемся этой темой в следующей главе.

Выбор элементов из итерируемых объектов: `filter`

Функция `map` – первоначальный и относительно прямолинейный представитель инструментального набора для функционального программирования в Python. Ее близкие родственники, `filter` и `reduce`, выбирают элементы из итерируемых объектов на основе проверочной функции и применяют функции к парам элементов соответственно.

Из-за возвращения итерируемого объекта функция `filter` (подобно `range`) требует вызова `list` при отображении всех ее результатов в Python 3.X. Скажем, следующий вызов `filter` выбирает из последовательности элементы больше нуля:

```
>>> list(range(-5, 5))           # Итерируемый объект в Python 3.X
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(filter(lambda x: x > 0, range(-5, 5)))      # Итерируемый объект
#   в Python 3.X
[1, 2, 3, 4]
```

Функция `filter` кратко упоминалась во врезке “Что потребует внимания: булевские операции” в конце главы 12 и во время исследования итерируемых объектов Python 3.X в главе 14. Элементы из последовательности либо итерируемого объекта, для которых проверочная функция возвращает истинное значение, добавляются в результирующий список. Как и `map`, функция `filter` приблизительно эквивалентна циклу `for`, но она является встроенной, лаконичной и часто более быстрой:

```
>>> res = []
>>> for x in range(-5, 5):          # Операторный эквивалент
    if x > 0:
        res.append(x)

>>> res
[1, 2, 3, 4]
```

Также подобно `map` функцию `filter` можно эмулировать посредством синтаксиса *спискового включения* с часто более простыми результатами (особенно если удается избежать создания новой функции) и с помощью похожего *генераторного выражения*, когда желателен отложенный выпуск результатов, но остаток истории мы рассмотрим в следующей главе:

```
>>> [x for x in range(-5, 5) if x > 0] # Используйте () для генерации элементов
[1, 2, 3, 4]
```

Комбинирование элементов из итерируемых объектов: `reduce`

Вызов функции `reduce`, которая в Python 2.X представляет собой встроенную функцию, но в Python 3.X находится в модуле `functools`, выглядит сложнее. Она принимает итерируемый объект, подлежащий обработке, но сама не является итерируемым объектом, а возвращает одиночный результат. Ниже показаны два вызова `reduce`, которые вычисляют сумму и произведение элементов в списке:

```
>>> from functools import reduce      # Импортируйте в Python 3.X,
#   но не в Python 2.X
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

На каждом шаге `reduce` передает текущую сумму или произведение вместе с очередным элементом из списка указанной функции `lambda`. По умолчанию начальное значение инициализируется первым элементом последовательности. В целях иллюстрации вот эквивалент в виде цикла `for` первого из двух вызовов с жестко закодированным сложением внутри цикла:

```
>>> L = [1, 2, 3, 4]
>>> res = L[0]
>>> for x in L[1:]:
    res = res + x

>>> res
10
```

Написание собственной версии `reduce` на самом деле довольно прямолинейно. Следующая функция эмулирует большую часть поведения встроенной функции и помогает прояснить ее работу в целом:

```
>>> def myreduce(function, sequence):
    tally = sequence[0]
    for next in sequence[1:]:
        tally = function(tally, next)
    return tally

>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])
15
>>> myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5])
120
```

Встроенная функция `reduce` разрешает передавать необязательный третий аргумент, который фактически помещается перед элементами последовательности и служит начальным значением, а также стандартным результатом, когда последовательность пуста, но мы оставим такое расширение для самостоятельного исследования.

Если продемонстрированная методика написания кода пробудила у вас интерес, тогда ваше внимание может привлечь стандартный библиотечный модуль `operator`, предлагающий функции, которые соответствуют встроенным выражениям и оказываются полезными в ряде сценариев использования инструментов функционального программирования (дополнительные сведения о модуле `operator` ищите в руководстве по библиотеке Python):

```
>>> import operator, functools
>>> functools.reduce(operator.add, [2, 4, 6])      # Операция +
# основанная
# на функции
12
>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])
12
```

Все вместе функции `map`, `filter` и `reduce` поддерживают мощные методики функционального программирования. Как уже упоминалось, многие эксперты склонны расширять инструментальный набор для функционального программирования в Python, дополняя его замыканиями вложенных областей видимости (известными еще и как фабричные функции) и анонимными функциями `lambda` (оба инструмента обсуждались ранее), а также *генераторами* и *включениями*, к которым мы снова обратимся в следующей главе.

Резюме

В главе были рассмотрены расширенные концепции, связанные с функциями: рекурсивные функции; аннотации функций; функции в форме выражений `lambda`; инструменты для функционального программирования, такие как `map`, `filter` и `reduce`; и общие принципы проектирования функций. В следующей главе мы продолжим тему расширенных возможностей исследованием генераторов и повторением итерируемых объектов и списковых включений – инструментов, которые имеют такое же отношение к функциональному программированию, как к операторам циклов. Однако прежде чем двигаться дальше, закрепите пройденный материал, ответив на традиционные контрольные вопросы по главе.

Проверьте свои знания: контрольные вопросы

1. Каким образом связаны друг с другом выражения `lambda` и операторы `def`?
2. Какой смысл применять `lambda`?
3. Сравните и противопоставьте `map`, `filter` и `reduce`.
4. Что такое аннотации функций и как их использовать?
5. Что такое рекурсивные функции и как их применять?
6. Назовите несколько универсальных принципов проектирования функций.
7. Назовите три или больше способов передачи результатов функций вызывающему коду.

Проверьте свои знания: ответы

1. Как `lambda`, так и `def` создает объект функции, подлежащий вызову в будущем. Тем не менее, поскольку `lambda` является выражением, оно возвращает объект функции, а не присваивает его имени, и может использоваться для вставки определения функции в места, где `def` синтаксически не допускается. Однако `lambda` разрешает неявно возвращать только одно выражение; из-за отсутствия поддержки блока операторов `lambda` не подходит для более крупных функций.
2. Выражения `lambda` позволяют “встраивать” небольших единиц исполняемого кода, откладывать их выполнение и снабжать их состоянием в форме стандартных аргументов и переменных из объемлющих областей видимости. Применение `lambda` не считается обязательным; вы всегда можете взамен написать `def` и ссылаться на функцию по имени. Тем не менее, выражения `lambda` оказываются полезными при встраивании небольших порций кода с отложенным выполнением, которые вряд ли будут использоваться в других местах программы. Они обычно встречаются в программах, основанных на обратных вызовах, таких как графические пользовательские интерфейсы, и имеют естественное сходство с инструментами функционального программирования вроде `map` и `filter`, которые ожидают обрабатывающую функцию.
3. Все три встроенные функции применяют другую функцию к элементам в последовательности (или в другом итерируемом объекте) и накапливают результаты. `map` передает каждый элемент функции и собирает все результаты, `filter`

накапливает элементы, для которых функция возвращает истинное значение, а `reduce` вычисляет единственное значение путем применения функции к накопителю и следующим друг за другом элементам. В отличие от остальных функций `reduce` в Python 3.X доступна внутри модуля `functools`, а не во встроенным контексте; `reduce` является встроенной функцией в Python 2.X.

4. Аннотации функций, доступные в Python 3.X (в Python 3.0 и последующих версиях), представляют собой синтаксические декораторы аргументов и результата функции, которые собираются в словарь, присваиваемый атрибуту `__annotations__` функции. Python не наделяет аннотации каким-либо смысловым значением, а просто упаковывает их для потенциального использования другими инструментами.
5. Рекурсивные функции вызывают сами себя либо прямо, либо косвенно с целью организации цикла. Они могут применяться для обхода структур произвольной формы и также для итерации вообще (хотя в последней роли часто проще и эффективнее использовать операторы цикла). Рекурсию нередко можно эмулировать или заменять кодом, в котором задействованы явные стеки или очереди для достижения большего контроля над обходами.
6. Функции, как правило, должны быть по возможности небольшими и самодостаточными, иметь единственное назначение и взаимодействовать с другими компонентами через входные аргументы и возвращаемые значения. Для передачи результатов они могут также применять изменяемые аргументы, если изменения ожидаемы, а некоторые типы программ заключают в себе другие механизмы передачи данных.
7. Функции могут отправлять обратно результаты с помощью операторов `return`, за счет модификации передаваемых изменяемых аргументов и путем установки глобальных переменных. Глобальные переменные, как правило, не одобряются (кроме особых случаев наподобие многопоточных программ), поскольку они могут сделать код более сложным для понимания и пользования. Обычно операторы `return` будут наилучшим вариантом, но модификация изменяемых аргументов тоже подходит (и даже удобна), если она ожидается. Функции также способны передавать результаты посредством системных механизмов, таких как файлы и сокеты, но это выходит за рамки рассматриваемых тем.

Включения и генераторы

В этой главе мы продолжим тему расширенных возможностей функций повторением концепций списковых включений и итерации, предварительный обзор которых был дан в главе 4, а формальное представление — в главе 14. Поскольку *включения* в той же степени связаны с рассмотренными в предыдущей главе инструментами *функционального программирования* (скажем, `map` и `filter`), как и с циклами `for`, здесь мы снова обратимся к ним в данном контексте. Мы также еще раз взглянем на итерируемые объекты с целью исследования *генераторных функций* и близкородственных им *генераторных выражений* — определяемых пользователем способов вырабатывания результатов по запросу.

Итерация в Python также охватывает определяемые пользователем *классы*, но мы отложим финал истории до части VI, где обсудим перегрузку операций. Однако с учетом того, что здесь завершается рассмотрение встроенных инструментов итерации, мы подведем итоги по разнообразным инструментам, встречавшимся до сих пор. В следующей главе будет показано, как измерять относительную производительность таких инструментов, в форме крупного учебного примера. А пока давайте продолжим тему включений и итерации, расширив ее генераторами значений.

Списковые включения и инструменты функционального программирования

Как упоминалось ранее в книге, Python поддерживает парадигмы процедурного, объектно-ориентированного и функционального программирования. На самом деле в Python имеется множество инструментов, которые большинство посчитают *функциональными* по своей природе; мы перечисляли их в предыдущей главе — замыкания, генераторы, лямбда-выражения, включения, отображения, декораторы, объекты функций и т.д. Такие инструменты позволяют эффективно применять и комбинировать функции, а также часто предлагают сохранение состояния и кодовые решения, которые являются альтернативами использованию классов и объектно-ориентированного программирования.

Например, в предыдущей главе рассматривались инструменты `map` и `filter` — главные члены раннего инструментального набора для функционального программирования на Python, появлению которых способствовал язык Lisp; они отображают операции на итерируемые объекты и накапливают результаты. Из-за того, что эта за-

дача настолько распространена при написании кода Python, в языке Python в итоге появилось новое выражение — *списковое включение*, которое обладает даже большей гибкостью, чем только что изученные инструменты.

Согласно истории Python вдохновением для ввода списковых включений поначалу послужил похожий инструмент в языке функционального программирования Haskell, что было примерно во времена версии Python 2.0. Вкратце списковые включения применяют к элементам в итерируемом объекте произвольное *выражение* вместо функции. Соответственно они могут быть более универсальными инструментами. В последних выпусках включения были расширены для исполнения других ролей — для множеств, словарей и даже выражений генераторов значений, которые мы исследуем в настоящей главе. Они больше не предназначены только для списков.

Впервые мы встретили списковые включения в предварительном обзоре в главе 4 и дополнительно обсуждали их в главе 14 в сочетании с операторами циклов. Тем не менее, поскольку они также относятся к инструментам функционального программирования вроде `map` и `filter`, мы еще раз возвращаемся к данной теме. Формально это средство не завязано на функции — мы увидим, что списковые включения способны быть более универсальным инструментом, чем `map` и `filter`, но иногда их проще понять по аналогии с альтернативами на базе функций.

Списковые включения или `map`

Давайте проработаем пример, который демонстрирует основы. Как было показано в главе 7,строенная функция `ord` в Python возвращает целочисленную кодовую точку одиночного символа (встроенная функция `chr` делает обратное — возвращает символ для целочисленной кодовой точки). Если символы попадают в диапазон 7-битных кодовых точек набора символов ASCII, тогда они будут кодами ASCII:

```
>>> ord('s')
115
```

Теперь предположим, что нужно собрать коды ASCII *всех* символов в единой строке. Возможно, наиболее прямолинейный подход предусматривает использование простого цикла `for` и добавление результатов в список:

```
>>> res = []
>>> for x in 'spam':
...     res.append(ord(x))           # Сбор результатов вручную
>>> res
[115, 112, 97, 109]
```

Однако, располагая сведениями о `map`, мы можем достичь похожих результатов без необходимости в управлении построением списка внутри кода:

```
>>> res = list(map(ord, 'spam'))    # Применение функции к последовательности
                                         # (или другому объекту)
>>> res
[115, 112, 97, 109]
```

Тем не менее, такие же результаты можно получить из выражения спискового включения — в то время как `map` отображает *функцию* на итерируемый объект, списковые включения отображают выражение на последовательность или другой итерируемый объект:

```
>>> res = [ord(x) for x in 'spam']  # Применение выражения к последовательности
                                         # (или другому объекту)
```

```
>>> res
[115, 112, 97, 109]
```

Списковые включения накапливают результаты применения произвольного выражения к итерируемому объекту, содержащему значения, и возвращают их в новом списке. Синтаксически списковые включения помещаются в квадратные скобки, чтобы напомнить о том, что они строят списки. В самой простой форме внутри квадратных скобок записывается выражение, указывающее переменную, за которой следует то, что выглядит похожим на заголовок цикла `for`, где задействована та же самая переменная. Затем Python накапливает результаты выражения для каждой итерации подразумеваемого цикла.

Эффект предыдущего примера аналогичен эффекту написанного вручную цикла `for` и вызова `map`. Однако списковые включения становятся более удобными, когда мы хотим применить к итерируемому объекту произвольное выражение, а не функцию:

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Здесь мы накапливаем квадраты чисел от 0 до 9 (мы просто позволяем интерактивной подсказке вывести результирующий списоковый объект; если хотите сохранить его, тогда присвойте какой-нибудь переменной). Чтобы выполнить подобную работу посредством вызова `map`, нам видимо потребуется создать небольшую функцию для операции возведения в квадрат. Из-за того, что эта функция больше нигде не нужна, она обычно (но не обязательно) записывается как внутристочная с помощью `lambda`, а не с использованием оператора `def` где-то в другом месте:

```
>>> list(map(lambda x: x ** 2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Код делает ту же самую работу и лишь немного длиннее эквивалентного спискового включения. Он также в самой минимальной степени сложнее (во всяком случае, если вы понимаете `lambda`). Но для более замысловатых разновидностей выражений списковые включения часто будут требовать гораздо меньшего объема набора. В следующем разделе показано почему.

Добавление проверок и вложенных циклов: `filter`

Списковые включения даже более универсальны, чем иллюстрировалось до сих пор. Скажем, как известно из главы 14, после `for` можно поместить конструкцию `if`, чтобы добавить логику выбора. Списковые включения с конструкциями `if` можно считать аналогом встроенной функции `filter`, обсуждаемой в предыдущей главе — они пропускают элементы итерируемого объекта, для которых конструкция `if` не дает истинное значение.

В целях демонстрации ниже показаны схемы выбора четных чисел от 0 до 4; подобно альтернативе `map` в виде спискового включения из предыдущего раздела версия с `filter` здесь должна создавать небольшую функцию `lambda` для проверочного выражения. Для сравнения также приведен эквивалентный цикл `for`:

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> list(filter((lambda x: x % 2 == 0), range(5)))
[0, 2, 4]

>>> res = []
>>> for x in range(5):
```

```
if x % 2 == 0:  
    res.append(x)  
  
>>> res  
[0, 2, 4]
```

Для обнаружения четных чисел все они используют операцию `%` деления по модулю (нахождения остатка от деления): если остаток от деления числа на 2 равен нулю, тогда число должно быть четным. Вызов `filter` также не намного длиннее, чем списковое включение. Тем не менее, в списковом включении мы можем скомбинировать конструкцию `if` и произвольное выражение, чтобы добиться эффекта `filter` и `map` в единственном выражении:

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]  
[0, 4, 16, 36, 64]
```

На этот раз мы накапливаем квадраты четных чисел от 0 до 9: цикл `for` пропускает числа, для которых присоединенная справа конструкция `if` дает ложное значение, и выражение слева вычисляет квадраты. Эквивалентный вызов `map` потребовал бы намного большей работы с нашей стороны — нам пришлось бы скомбинировать выбор с помощью `filter` и итерацию посредством `map`, создавая заметно более сложное выражение:

```
>>> list( map(lambda x: x**2), filter(lambda x: x % 2 == 0, range(10)))  
[0, 4, 16, 36, 64]
```

Формальный синтаксис включений

В действительности списковые включения даже более универсальны. Их простейшая форма предусматривает указание накапливающего выражения и одиночной конструкции `for`:

```
[ выражение for цель in итерируемый_объект ]
```

Несмотря на необязательность всех остальных частей, они позволяют выражать более развитые итерации — в списковом включении допускается записывать любое количество вложенных циклов `for`, каждое из которых может иметь необязательную ассоциированную проверку `if`, действующую в качестве фильтра. Общая структура списковых включений выглядит следующим образом:

```
[ выражение for цель1 in итерируемый_объект1 if условие1  
      for цель2 in итерируемый_объект2 if условие2 ...  
      for цельN in итерируемый_объектN if условиеN ]
```

Такой же синтаксис унаследован включениями *множеств* и *словарей*, а также появившимися позже *генераторными выражениями*, хотя они используют другие объемлющие символы (фигурные скобки или часто необязательные круглые скобки), а включение словаря начинается с двух выражений, разделенных двоеточием (для ключа и значения).

Мы экспериментировали с конструкцией фильтрации `if` в предыдущем разделе. Когда конструкции `for` вкладываются внутри спискового включения, они работают подобно эквивалентным вложенным операторам цикла `for`. Вот пример:

```
>>> res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]  
>>> res  
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Эффект будет таким же, как у показанного ниже существенно более многословного эквивалента:

```
>>> res = []
>>> for x in [0, 1, 2]:
    for y in [100, 200, 300]:
        res.append(x + y)

>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Хотя списковые включения создают результирующие списки, не забывайте, что они способны проходить по любой последовательности или другому итерируемому объекту. Далее приведен похожий фрагмент кода, который вместо списков чисел обходит строки, накапливая результаты конкатенации:

```
>>> [x + y for x in 'spam' for y in 'SPAM']
['sS', 'sP', 'sA', 'sM', 'pS', 'pP', 'pA', 'pM',
 'aS', 'aP', 'aA', 'aM', 'mS', 'mP', 'mA', 'mM']
```

Каждая конструкция `for` может иметь ассоциированный фильтр `if` независимо от того, насколько глубоко вложены циклы — хотя сценарии использования для кода следующего вида, не считая многоуровневых массивов, придумывать все труднее и труднее:

```
>>> [x + y for x in 'spam' if x in 'sm' for y in 'SPAM' if y in ('P', 'A')]
['sP', 'sA', 'mP', 'mA']

>>> [x + y + z for x in 'spam' if x in 'sm'
      for y in 'SPAM' if y in ('P', 'A')
      for z in '123' if z > '1']
['sP2', 'sP3', 'sA2', 'sA3', 'mP2', 'mP3', 'mA2', 'mA3']
```

Наконец, ниже представлено аналогичное списковое включение, которое иллюстрирует эффект от присоединения конструкций выбора `if` к вложенным конструкциям `for`, применяемым к числовым объектам вместо строк:

```
>>> [(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Выражение комбинирует четные числа от 0 до 4 с нечетными числами от 0 до 4. Конструкции `if` отфильтровывают элементы на каждой итерации. Вот эквивалентный код на основе операторов:

```
>>> res = []
>>> for x in range(5):
    if x % 2 == 0:
        for y in range(5):
            if y % 2 == 1:
                res.append((x, y))

>>> res
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Вспомните, что если вам не вполне ясна работа сложного спискового включения, то вы всегда можете подобным образом вложить конструкции `for` и `if` спискового включения внутрь друг друга (последовательно смешая вправо каждую конструкцию) и получить эквивалентные операторы. Результат окажется длиннее, но возможно на первый взгляд будет яснее в плане намерений для некоторых читателей, особенно когда они лучше знакомы с базовыми операторами.

Эквивалентные версии со встроенными функциями `map` и `filter` последнего примера были бы непомерно сложными и глубоко вложенными, поэтому я даже не

пытаюсь воспроизвести их здесь, а оставляю в качестве упражнения мастерам дзен, бывшим программистам на Lisp и попросту безрассудным!

Пример: списковые включения и матрицы

Разумеется, не все списковые включения настолько искусственны. Давайте взглянем на еще одно приложение, чтобы поразмять свои синапсы. Как было показано в главах 4 и 8, один из основных способов записи матриц (многомерных массивов) в Python задействует вложенные списковые структуры. Например, в следующем коде определяются две матрицы 3×3 в виде списков вложенных списков:

```
>>> M = [[1, 2, 3],  
        [4, 5, 6],  
        [7, 8, 9]]  
  
>>> N = [[2, 2, 2],  
        [3, 3, 3],  
        [4, 4, 4]]
```

Имея такую структуру, мы всегда можем индексировать по строкам и столбцам внутри строк с использованием нормальных операций индексации:

```
>>> M[1]          # Стока 2  
[4, 5, 6]  
  
>>> M[1][2]      # Стока 2, элемент 3  
6
```

Однако списковые включения являются мощным инструментом для обработки таких структур, потому что они автоматически просматривают строки и столбцы. Скажем, хотя такая структура хранит матрицу по строкам, для сбора значений *второго* столбца мы можем просто проходить по строкам и извлекать желаемый столбец или проходить по позициям в строках с индексацией в ходе дела:

```
>>> [row[1] for row in M]           # Столбец 2  
[2, 5, 8]  
  
>>> [M[row][1] for row in (0, 1, 2)]    # Использование смещений  
[2, 5, 8]
```

По заданным позициям мы также можем выполнять задачи наподобие извлечения *диагонали*. В первом из приведенных далее выражений с применением функции `range` генерируется список смещений и производится индексация с одинаковыми номерами строк и столбцов, выбирающая $M[0][0]$, затем $M[1][1]$ и т.д. Во второй строке индекс столбца уравновешивается для извлечения $M[0][2]$, $M[1][1]$ и т.д. (мы предполагаем, что матрица имеет равное количество строк и столбцов):

```
>>> [M[i][i] for i in range(len(M))]    # Диагонали  
[1, 5, 9]  
  
>>> [M[i][len(M)-1-i] for i in range(len(M))]  
[3, 5, 7]
```

Изменение такой матрицы *на месте* требует присваивания по смещениям (если форма отличается, тогда используйте `range` дважды):

```
>>> L = [[1, 2, 3], [4, 5, 6]]  
>>> for i in range(len(L)):  
    for j in range(len(L[i])):    # Обновление на месте  
        L[i][j] += 10
```

```
>>> L
[[11, 12, 13], [14, 15, 16]]
```

В действительности мы не в состоянии сделать то же самое с помощью списковых включений, т.к. они создают *новые списки*, но всегда могли бы присвоить их результаты исходному имени, добившись похожего эффекта. Например, мы можем применить операцию к каждому элементу в матрице, производя результаты в простом векторе или матрице той же формы:

```
>>> [col + 10 for row in M for col in row]      # Присваивание M для сохранения
                                                # нового значения
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> [[col + 10 for col in row] for row in M]
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

Чтобы понять такой код, его следует преобразовать к виду простых операторов с отступами частей, находящихся правее в выражении (как в первом цикле ниже), и создать новый список, когда включения вкладываются слева (подобно второму циклу ниже). Как прояснил операторный эквивалент, второе выражение в предыдущем коде работает из-за того, что итерация по строкам является внешним циклом — для каждой строки она запускает вложенную итерацию по столбцам, чтобы построить одну строку результирующей матрицы:

```
>>> res = []
>>> for row in M:          # Эквиваленты в виде операторов
    for col in row:        # Отступ для частей, находящихся дальше справа
        res.append(col + 10)

>>> res
[11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> res = []
>>> for row in M:
    tmp = []               # Вложенное слева включение начинает новый список
    for col in row:
        tmp.append(col + 10)
    res.append(tmp)

>>> res
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

Наконец, проявив толику творческого подхода, мы также можем использовать списковые включения для комбинирования значений *множества матриц*. В показанном далее коде сначала строится плоский список, который содержит результат попарного перемножения матриц, а затем структура вложенного списка, имеющая те же самые значения, снова за счет вложения списковых включений:

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> N
[[2, 2, 2], [3, 3, 3], [4, 4, 4]]
>>> [M[row][col] * N[row][col] for row in range(3) for col in range(3)]
[2, 4, 6, 12, 15, 18, 28, 32, 36]
>>> [[M[row][col] * N[row][col] for col in range(3)] for row in range(3)]
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

И опять последнее выражение работает из-за того, что итерация по строкам является внешним циклом; оно эквивалентно следующему коду на основе операторов:

```
res = []
for row in range(3):
    tmp = []
    for col in range(3):
        tmp.append(M[row][col] * N[row][col])
    res.append(tmp)
```

Ради еще большей забавы мы можем применить `zip` для образования пар элементов с целью перемножения. Приведенные ниже версии со списковым включением и операторами цикла производят тот же самый список списков с результатами попарного перемножения, что и предшествующий пример (и поскольку `zip` в Python 3.X представляет собой генератор значений, решение не настолько неэффективно, как может показаться):

```
[[col1 * col2 for (col1, col2) in zip(row1, row2)] for (row1, row2) in zip(M, N)]
res = []
for (row1, row2) in zip(M, N):
    tmp = []
    for (col1, col2) in zip(row1, row2):
        tmp.append(col1 * col2)
    res.append(tmp)
```

По сравнению с эквивалентными операторами версия со списковым включением требует только одной строки кода, может выполняться значительно быстрее для крупных матриц и просто способна взорвать ваш мозг! Что и подводит нас к следующему разделу.

Не злоупотребляйте списковыми включениями: KISS

При такой универсальности списковые включения могут быстро стать малопонятными, особенно когда они вложенные. Некоторые задачи программирования сложны по своей природе, и мы не можем приукрашивать их в попытке сделать проще, чем они есть на самом деле (в качестве основного примера дождитесь грядущих перестановок). Инструменты вроде включений являются мощными решениями, когда используются разумно, и по существу нет ничего плохого в том, чтобы применять их в своих сценариях.

Вместе с тем код, подобный представленному в предыдущем разделе, может выйти за границы допустимой сложности — и, откровенно говоря, способен непропорционально возбудить интерес у тех, кто придерживается ошибочного предположения о том, что запутывание кода каким-то образом намекает на наличие таланта. Поскольку такие инструменты имеют тенденцию быть притягательными для определенных людей больше, чем вероятно должны, я вынужден прояснить здесь область их применения.

В книге демонстрируются усложненные включения в учебных целях, но в реальном мире использование неоправданно трудного для понимания и замысловатого кода служит признаком скверного проектирования и плохой заботы о его потребителях. Перефразируя мысль из первой главы: программа не должна быть заумной и непонятной — она должна четко сообщать о своем предназначении.

Или вот один из девизов, выводимых `import this`:

Простое лучше сложного.

Написание кода со сложными включениями может быть веселым академическим развлечением, но ему не место в программах, которые когда-нибудь придется разбирать другим.

Поэтому вот вам мой совет: в начале работы с Python применяйте простые циклы `for` и включения или вызовы `map` в отдельных случаях, где их легко использовать. Как всегда, здесь применимо правило “сохраняйте его простым”: лаконичность кода – гораздо менее важная цель, чем его читабельность. Если вы вынуждены переводить код в операторы, чтобы понять его, то вероятно там и должны быть операторы. Таким образом, старый акроним *KISS* по-прежнему актуален: фраза *Keep It Simple* (сохраняйте его простым), за которой следует либо слово, считающееся в наши дни слишком сексистским (*Sir* (сэр)), либо чересчур красочным для такой книги, читаемой всей семьей...

Обратная сторона: производительность, лаконичность, выразительность

Тем не менее, применительно к данному случаю в настоящее время дополнительная сложность дает солидное преимущество в плане производительности: современные тесты в Python показывают, что вызовы `map` могут выполняться вдвое быстрее эквивалентных циклов `for`, а списковые включения зачастую даже быстрее вызовов `map`. Разница в скорости может варьироваться в зависимости от особенностей использования и версии Python, но в целом объясняется тем фактом, что `map` и списковые включения выполняются со скоростью кода на языке С внутри интерпретатора, которая часто гораздо выше, чем скорость выполнения байт-кода циклов `for` внутри PVM.

Добавок списковые включения обеспечивают **лаконичность** кода, которая поощряется и даже предписывается, когда из такого сокращения размера не вытекает также уменьшение смысла для следующего программиста. Кроме того, многие считают **выразительность** включений мощным союзником. Поскольку вызовы `map` и списковые включения представляют собой выражения, синтаксически они также разрешены в местах, где операторы цикла `for` не допускаются, скажем, в теле функций `lambda`, внутри списковых и словарных литералов и т.п.

По указанным причинам списковые включения и вызовы `map` заслуживают изучения и применения для более простых видов итераций, особенно если скорость приложения является важным фактором. Однако из-за того, что циклы `for` делают логику более явной, они в целом рекомендуются на основании простоты и зачастую для получения более прямолинейного кода. В случае использования вы должны стараться сохранять вызовы `map` и списковые включения простыми; для более сложных задач взамен применяйте полные операторы.



Как уже упоминалось, выигрыш в *производительности* зависит от моделей вызовов, а также изменений и оптимизаций в самом Python. Например, в недавних выпусках Python было ускорено выполнение простого оператора цикла `for`. Тем не менее, в определенном коде списковые включения по-прежнему выполняются значительно быстрее циклов `for` и даже вызовов `map`, хотя функция `map` все еще может выигрывать, когда альтернативные варианты должны применять вызов функции – встроенной или какой-то другой. До тех пор, пока ситуация своеобразно не изменится, самостоятельно измеряйте время выполнения альтернативных реализаций с помощью инструментов в стандартном библиотечном модуле `time` или более новом модуле `timeit`, появившемся в версии Python 2.4. В следующей главе мы подробно обсудим оба модуля и докажем приведенные выше утверждения.

Что потребует внимания: списковые включения и map

Давайте рассмотрим несколько более реалистичных примеров списковых включений и вызовов map в действии. Первый пример мы решали в главе 14 с использованием списковых включений, но здесь мы исследуем альтернативную версию с применением map. Вспомните, что файловый метод readlines возвращает строки с символами \n в конце (в следующем коде предполагается наличие 3-строчного текстового файла в текущем каталоге):

```
>>> open('myfile').readlines()
['aaa\n', 'bbb\n', 'ccc\n']
```

Если вам не нужны символы \n, тогда можете отсечь их во всех строках за один шаг с помощью спискового включения или вызова map (результаты map являются итерируемыми объектами в Python 3.X, поэтому их необходимо прогнать через вызов list, чтобы отобразить сразу все результаты):

```
>>> [line.rstrip() for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']
>>> [line.rstrip() for line in open('myfile')]
['aaa', 'bbb', 'ccc']
>>> list(map((lambda line: line.rstrip()), open('myfile')))
['aaa', 'bbb', 'ccc']
```

В последних двух версиях используются *файловые итераторы*; как было показано в главе 14, это означает, что вам не понадобится вызывать какой-то метод для чтения строк в итерационных контекстах. Вызов map чуть длиннее спискового включения, но в обоих случаях нет необходимости в явном управлении созданием результирующего списка.

Списковое включение может применяться также как разновидность операции проецирования столбцов. Стандартный API-интерфейс для работы с базами данных SQL в Python возвращает результаты запросов в форме последовательности последовательностей вроде следующей ниже; список представляет таблицу, кортежи — строки, а элементы в кортежах — значения столбцов:

```
>>> listoftuple = [('bob', 35, 'mgr'), ('sue', 40, 'dev')]
```

С использованием цикла for можно было бы извлечь все значения из выбранного столбца вручную, но вызовы map и списковые включения способны выполнить такую работу за один шаг, к тому же быстрее:

```
>>> [age for (name, age, job) in listoftuple]
[35, 40]
>>> list(map((lambda row: row[1]), listoftuple))
[35, 40]
```

В первом варианте задействовано *присваивание кортежа*, чтобы распаковать кортежи со строками в списке, а во втором применяется индексация. В Python 2.X (но не в Python 3.X — см. врезку “На заметку!” о распаковке аргументов в Python 2.X, приведенную в главе 18) функция map тоже может использовать распаковку кортежей в своем аргументе:

```
# Только Python 2.X
>>> list(map((lambda (name, age, job): age), listoftuple))
[35, 40]
```

Дополнительные сведения относительно API-интерфейса для работы с базами данных в Python ищите в других книгах и ресурсах.

Помимо различий в выполнении функций и выражений самое большое несходство между map и списковыми включениями в Python 3.X связано с тем, что map является *итерируемым объектом*, генерирующим результаты по запросу. Чтобы достичь той же экономии памяти и времененного разделения при выполнении, списковые включения должны записываться как *генераторные выражения* — одна из крупных тем этой главы, которая рассматривается следующей.

Генераторные функции и выражения

В наши дни Python поддерживает откладывание гораздо больше, чем было в прошлом – он предоставляет инструменты, которые производят результаты, только когда необходимо, а не все одновременно. Мы видели откладывание при работе встроенных инструментов: файлов, которые читают строки по запросу, и функции наподобие `map` и `zip`, производящие элементы по требованию в Python 3.X. Однако такой ленивый подход не ограничивается самим Python. В частности, есть две языковые конструкции, которые откладывают создание результатов всякий раз, когда это возможно, в определяемых пользователем операциях.

- Генераторные функции (доступные, начиная с версии Python 2.3) записываются как нормальные операторы `def`, но в них применяются операторы `yield`, чтобы возвращать по одному результату за раз, приостанавливать выполнение с сохранением состояния и возобновлять его между выдачами.
- Генераторные выражения (доступные, начиная с версии Python 2.4) похожи на списковые включения, описанные в предыдущем разделе, но они не строят результирующий список, а возвращают объекты, которые производят результаты по запросу.

Из-за того, что ни та, ни другая конструкция не создает сразу весь результирующий список, они экономят пространство памяти и позволяют распределить время вычислений по запросам результатов. Как мы увидим, обе конструкции в конечном итоге делают свою магию отложенных результатов за счет реализации *протокола итерации*, который обсуждался в главе 14.

Указанные средства не новы (генераторные функции были доступны на выбор раньше Python 2.2) и в наши дни довольно распространены в коде Python. Понятие генераторов Python многим обязано другим языкам программирования, особенно Icon. Хотя поначалу они могут показаться необычными в случае, если вы привыкли к более простым программным моделям, в подходящих ситуациях вы наверняка сочтете генераторы мощным инструментом. Кроме того, поскольку они являются естественным расширением исследованных ранее идей функций, включений и итерации, вы уже знаете намного больше о написании генераторов, чем могли бы ожидать.

Генераторные функции: `yield` или `return`

В этой части книги мы выяснили, как писать код нормальных функций, которые принимают входные параметры и немедленно отправляют обратно одиночный результат. Тем не менее, можно также писать функции, которые способны отправлять обратно значение и позже возобновлять работу с места, которое они оставили. Такие функции, доступные в Python 2.X и 3.X, известны как *генераторные функции*, потому что они генерируют последовательность значений с течением времени.

Генераторные функции во многих отношениях похожи на нормальные функции и фактически записываются с помощью обычных операторов. Однако при их создании они специально компилируются в объект, который поддерживает протокол итерации. Когда генераторные функции вызываются, они возвращают не результат, а генератор результатов, который может появляться в любом итерационном контексте. Мы исследовали итерируемые объекты в главе 14, и на рис. 14.1 представили формальный и графический обзор их работы. Здесь мы снова к ним вернемся, чтобы посмотреть, как они связаны с генераторами.

Приостановка выполнения с сохранением состояния

В отличие от нормальных функций, которые возвращают значение и прекращают работу, генераторные функции автоматически приостанавливают и возобновляют свое выполнение и состояние вокруг точки генерации значений. По этой причине они часто будут служить удобной альтернативой заблаговременному вычислению полной серии значений с последующим сохранением и восстановлением вручную состояния посредством классов. *Состояние*, которое генераторные функции сохраняют, когда приостанавливаются, содержит местоположение в коде и полную локальную область видимости. Следовательно, их локальные переменные помнят информацию между выпуском результатов и делают ее доступной при возобновлении выполнения функциями.

Основная разница между кодом генераторных и нормальных функций заключается в том, что генератор *выдает* значение, а не *возвращает* его – оператор `yield` приостанавливает функцию и отправляет значение обратно вызывающему коду, но сохраняет достаточный объем состояния, чтобы предоставить функции возможность возобновить работу с места, которое она покинула. При возобновлении функция продолжает выполнение сразу после последнего запуска `yield`. С точки зрения функции такой прием позволяет ее коду производить серию значений с течением времени вместо вычисления их всех сразу и возвращения в чем-нибудь, подобном списку.

Объединение с протоколом итерации

Чтобы по-настоящему понять генераторные функции, вы должны знать, что они тесно связаны с понятием протокола итерации в Python. Как вы видели, объекты итераторов определяют метод `__next__` (`next` в Python 2.X), который либо возвращает очередной элемент в итерации, либо инициирует специальное исключение `StopIteration` для окончания итерации. Итератор итерируемого объекта первоначально извлекается с помощью встроенной функции `iter`, хотя этот шаг ничего не делает для объектов, которые сами являются итераторами.

Циклы `for` в Python и все остальные итерационные контексты используют протокол итерации для прохода по генератору последовательностей или значений, если протокол поддерживается (если нет, тогда итерация взамен предпринимает много-кратную индексацию последовательностей). Любой объект, поддерживающий такой интерфейс, работает со всеми итерационными инструментами.

Для поддержки протокола итерации функции, содержащие оператор `yield`, компилируются особым образом как *генераторы* – они не будут нормальными функциями, а строятся с целью возвращения объекта с ожидаемыми методами из протокола итерации. При последующем вызове они возвращают объект генератора, который поддерживает интерфейс итерации с автоматически созданным методом по имени `__next__`, предназначенный для запуска или возобновления выполнения.

Генераторные функции могут также иметь оператор `return`, который наряду с перемещением за конец блока `def` просто прекращает генерацию значений – формально за счет инициирования исключения `StopIteration` после всех обычных действий по выходу из функции. С точки зрения вызывающего кода метод `__next__` генератора возобновляет выполнение функции до тех пор, пока либо не возвратится следующий результат `yield`, либо до возникновения исключения `StopIteration`.

Совокупный эффект в том, что генераторные функции, которые записаны как операторы `def`, содержащие операторы `yield`, автоматически делаются доступными для протокола итерации и потому могут применяться в любом итерационном контексте, чтобы производить результаты с течением времени и по запросу.



Как отмечалось в главе 14, в Python 2.X объекты итераторов определяют метод по имени `next`, а не `__next__`. Это касается используемых здесь объектов генераторов. В Python 3.X данный метод переименован в `__next__`. Встроенная функция `next` предоставляется как удобный и переносимый инструмент: `next(I)` – то же самое, что `I.__next__()` в Python 3.X и `I.next()` в Python 2.6/2.7. До версии Python 2.6 в программах вместо ручной итерации просто вызывался `I.next()`.

Генераторные функции в действии

Для иллюстрации основ генераторов давайте перейдем к написанию кода. В следующем коде определяется генераторная функция, которую можно применять для генерации квадратов серии чисел с течением времени:

```
>>> def gensquares(N):
    for i in range(N):
        yield i ** 2    # Позже возобновить здесь выполнение
```

Функция `gensquares` выдает значение и потому возвращает управление вызывающему коду на каждой итерации цикла; при возобновлении ее выполнения восстанавливается предыдущее состояние, включая последние значения переменных `i` и `N`, а управление снова подхватывается непосредственно после оператора `yield`. Скажем, когда `gensquares` используется в теле цикла `for`, первая итерация начинает функцию и получает первый результат; затем на каждой итерации цикла управление возвращается функции после оператора `yield`:

```
>>> for i in gensquares(5):      # Возобновление выполнения функции
    print(i, end=' ')           # Вывод последнего выданного значения
0 : 1 : 4 : 9 : 16 :
```

Чтобы закончить генерацию значений, функции либо применяют оператор `return` без значения, либо позволяют потоку управления дойти до конца тела функции¹.

Большинству людей такой процесс на первый взгляд кажется не совсем явным (а то и вообще магическим). Но на самом деле он вполне материщен. Если вы действительно хотите увидеть, что происходит внутри `for`, тогда вызовите генераторную функцию напрямую:

```
>>> x = gensquares(4)
>>> x
<generator object gensquares at 0x000000000292CA68>
```

Вы получите обратно объект генератора, который поддерживает протокол итерации, встречавшийся в главе 14 – генераторная функция была скомпилирована так, чтобы возвращать его автоматически. В свою очередь возвращенный объект генератора имеет метод `__next__`, который запускает функцию или возобновляет ее выполнение с места, откуда она последний раз выдала значение, и инициирует исключение `StopIteration`, когда достигнут конец серии значений и происходит возврат управления из функции.

¹ Формально Python трактует наличие значения в операторе `return` внутри генераторных функций как синтаксическую ошибку в Python 2.X и во всех версиях Python 3.X, предшествующих Python 3.3. Начиная с версии Python 3.3, значение в операторе `return` разрешено и присоединяется к объекту `StopIteration`, но оно игнорируется в контекстах автоматических итераций, а его использование делает код несовместимым со всеми предыдущими выпусками.

Для удобства встроенная функция `next(X)` вызывает метод `X.__next__()` объекта в Python 3.X (и `X.next()` в Python 2.X):

```
>>> next(x)      # То же, что и x.__next__() в Python 3.X
0
>>> next(x)      # Используйте x.next() или next() в Python 2.X
1
>>> next(x)
4
>>> next(x)
9
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
StopIteration
```

Как выяснилось в главе 14, циклы `for` (и другие итерационные контексты) работают с генераторами одинаково – многократно вызывая метод `__next__`, пока не возникнет исключение. Для генератора результатом является производство выдаваемых значений с течением времени. Если подлежащий итерации объект не поддерживает такой протокол, то циклы `for` взамен применяют протокол индексирования.

Обратите внимание, что вызов `iter` верхнего уровня в протоколе итерации здесь необязателен, поскольку генераторы сами являются итераторами, поддерживающими только один активный просмотр во время итерации. Говоря по-другому, генераторы возвращают для `iter` самих себя, т.к. они поддерживают `next` напрямую. Сказанное также остается справедливым для генераторных выражений, которые мы обсудим позже в главе:

```
>>> y = gensquares(5)      # Возвращает генератор, который сам по себе является
итератором
>>> iter(y) is y          # iter() не требуется: здесь ничего не делает
True
>>> next(y)                # Можно запускать next() непосредственно
0
```

Для чего используются генераторные функции?

С учетом простых примеров, применяемых для иллюстрации основ, вас может интересовать, зачем вообще писать код генератора. Скажем, в примере текущего раздела можно было бы просто построить сразу весь список выданных значений:

```
>>> def buildsquares(n):
    res = []
    for i in range(n): res.append(i ** 2)
    return res

>>> for x in buildsquares(5): print(x, end=' : ')
0 : 1 : 4 : 9 : 16 :
```

По существу мы могли бы использовать любую методику с циклом `for`, функцией `map` или списковым включением:

```
>>> for x in [n ** 2 for n in range(5)]:
    print(x, end=' : ')

0 : 1 : 4 : 9 : 16 :
```

```
>>> for x in map((lambda n: n ** 2), range(5)):
    print(x, end=' : ')
0 : 1 : 4 : 9 : 16 :
```

Тем не менее, в крупных программах генераторы могут быть лучше в плане памяти и производительности. Они позволяют функциям избежать выполнения всей работы заранее, что особенно полезно, когда результирующие списки большие или получение каждого значения требует длительных вычислений. Генераторы распределяют время, необходимое для производства серии значений, по всем итерациям цикла.

Кроме того, для усложненных сценариев применения генераторы способны предложить более простую альтернативу ручному сохранению состояния между итерациями в объектах классов — генераторы обеспечивают автоматическое сохранение и восстановление переменных, доступных в области видимости функций². Мы более подробно обсудим итерируемые объекты на основе классов в части VI.

Генераторные функции гораздо более универсальны, чем могли показаться. Они способны действовать и возвращать объекты любых типов, а также как итерируемые объекты могут появляться во всех итерационных контекстах из главы 14, в том числе в вызовах `tuple`, перечислениях и включениях словарей:

```
>>> def ups(line):
    for sub in line.split(','):# Генератор подстрок
        yield sub.upper()
>>> tuple(ups('aaa,bbb,ccc'))# Все итерационные контексты
('AAA', 'BBB', 'CCC')
>>> {i: s for (i, s) in enumerate(ups('aaa,bbb,ccc'))}
{0: 'AAA', 1: 'BBB', 2: 'CCC'}
```

Вскоре мы увидим те же самые ресурсы для генераторных выражений — инструмента, который обменивает гибкость функций на лаконичность включений. Позже в главе мы также выясним, что временами генераторы делают невозможное возможным, производя компоненты результирующих наборов, которые оказались бы слишком крупными, чтобы быть созданными полностью за раз. Но сначала давайте исследуем ряд более развитых возможностей генераторных функций.

Расширенный протокол генераторных функций: `send` или `next`

В версии Python 2.5 к протоколу генераторных функций был добавлен метод `send`. Он осуществляет переход на следующий элемент в серии результатов, в частности как `__next__`, но также снабжает вызывающий код возможностью взаимодействия с генератором для влияния на его работу.

² Интересно отметить, что генераторные функции также являются чем-то вроде механизма *многопоточной обработки* “для бедняков” — они чередуют работу функции с работой вызывающего кода за счет разделения действия функции на шаги, выполняемые между вызовами `yield`. Однако генераторы — не потоки: программа явно направляется на функцию и из нее внутри единственного потока управления. Многопоточная обработка в каком-то смысле более универсальна (производители результатов способны выполнять по-настоящему независимо и отправлять результаты в очередь), но генераторы могут быть проще в написании. Краткое введение в инструменты многопоточной обработки Python ищите в сноске, посвященной многопоточности, в главе 17. Обратите внимание, что поскольку управление явно направляется на вызовы `yield` и `next`, генераторы также не *возвращаются к предыдущему состоянию* и имеют более тесное отношение к *сопрограммам* — формальные понятия, которые выходят за рамки тематики настоящей главы.

Формально `yield` теперь имеет форму не оператора, а выражения, которое возвращает элемент, переданный `send` (хотя `yield` можно вызывать любым из двух способов – `yield X` или `A = (yield X)`). Выражение должно заключаться в круглые скобки, если только оно не является единственным элементом в правой части оператора присваивания, например, `X = yield Y` и `X = (yield Y) + 42`.

Когда используется такой расширенный протокол, значения отправляются генератору `G` путем вызова `G.send(значение)`. Затем код генератора возобновляет выполнение, и выражение `yield` в генераторе возвращает значение, переданное `send`. Если вызывается обычный метод `G.__next__()` (или его эквивалент `next(G)`) для продвижения вперед, тогда `yield` просто возвращает `None`. Вот пример:

```
>>> def gen():
    for i in range(10):
        X = yield i
    print(X)

>>> G = gen()
>>> next(G)      # Сначала должен вызываться next(), чтобы запустить генератор
0
>>> G.send(77)   # Продвигается вперед и отправляет значение выражению yield
77
1
>>> G.send(88)
88
2
>>> next(G)      # next() и X.__next__() отправляют None
None
3
```

Метод `send` можно применять, например, для написания генератора, который разрешает прекращать свою работу за счет отправки кода завершения либо изменять направление путем передачи новой позиции в данных, обрабатываемых внутри генератора.

Добавок генераторы в Python 2.5 и последующих версиях также поддерживают метод `throw(тип)`, предназначенный для инициирования исключения внутри генератора в самом последнем `yield`, и метод `close`, который инициирует специальное исключение `GeneratorExit` внутри генератора, чтобы полностью прекратить итерацию. Указанные методы представляют собой расширенные средства, которые здесь подробно не рассматриваются; за дополнительными сведениями обращайтесь к справочникам и стандартным руководствам по Python, а сами исключения детально обсуждаются в части VII.

Обратите внимание, что в то время, как Python 3.X предлагает удобную встроенную функцию `next(X)`, которая вызывает метод `X.__next__()` объекта, другие методы генераторов наподобие `send` должны вызываться напрямую как методы объектов генераторов (скажем, `G.send(X)`). Смысл станет понятным, если вы осознаете, что эти добавочные методы реализованы только во встроенных объектах генераторов, тогда как метод `__next__` применяется ко всем итерируемым объектам – встроенных типов или определяемых пользователем классов.

Также следует отметить, что в Python 3.3 было введено расширение `yield` – конструкция `from`, которая позволяет генераторам делегировать работу вложенным генераторам. Поскольку это расширение темы, которая сама по себе довольно сложная, мы вынесем ее в отдельную врезку и зайдемся инструментом, достаточно близким, чтобы называться двойником.

Генераторные выражения: итерируемые объекты встречаются с включениями

Из-за того, что отложенная оценка генераторных функций оказалась настолько удобной, со временем она распространилась на другие инструменты. В Python 2.X и 3.X понятия итерируемых объектов и списковых включений были объединены в новый инструмент — *генераторные выражения*. Синтаксически генераторные выражения похожи на нормальные списковые включения и поддерживают весь их синтаксис, в том числе фильтры *if* и вложение циклов, но они помещаются в круглые скобки, а не в квадратные (подобно кортежам объемлющие круглые скобки часто необязательны):

```
>>> [x ** 2 for x in range(4)]      # Списковое включение: строит список  
[0, 1, 4, 9]  
>>> (x ** 2 for x in range(4))    # Генераторное выражение:  
#     создает итерируемый объект  
<generator object <genexpr> at 0x000000000029A8288>
```

На самом деле, во всяком случае, с точки зрения функциональности, написание кода спискового включения по существу будет тем же, что и помещение генераторного выражения в вызов встроенной функции *list* для принудительного выработывания всех результатов сразу:

```
>>> list(x ** 2 for x in range(4))    # Эквивалентность списковому включению  
[0, 1, 4, 9]
```

Тем не менее, генераторные выражения совершенно другие: вместо построения результирующего списка в памяти они возвращают *объект генератора* — автоматически созданный итерируемый объект. В свою очередь этот итерируемый объект поддерживает *протокол итерации*, чтобы выдавать по одной порции результирующего списка за раз в любом итерационном контексте. Итерируемый объект также сохраняет состояние генератора, пока он действует — переменную *x* в предшествующих выражениях наряду с местоположением в коде генератора.

Совокупный эффект во многом подобен генераторным функциям, но в контексте *выражения включения*: мы получаем обратно объект, который запоминает оставленное им место после возвращения каждой части результата. Как и в случае генераторных функций, исследование внутренностей протокола, который такие объекты автоматически поддерживают, может помочь прояснить их; здесь снова нет необходимости в вызове *iter* по причинам, объясняемым далее:

```
>>> G = (x ** 2 for x in range(4))  
>>> iter(G) is G      # Вызов iter(G) необязателен: __iter__ возвращает сам объект  
True  
>>> next(G)         # Объекты генераторов: автоматические методы  
0  
>>> next(G)  
1  
>>> next(G)  
4  
>>> next(G)  
9  
>>> next(G)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
Трассировка (самый последний вызов указан последним):  
  Файл <stdin>, строка 1, в <модуль>
```

```
StopIteration
>>> G
<generator object <genexpr> at 0x00000000029A8318>
```

Опять-таки обычно мы не видим внутренний итерационный механизм `next` генераторного выражения вроде показанного, т.к. циклы `for` запускают его автоматически:

```
>>> for num in (x ** 2 for x in range(4)):    # Автоматически вызывает next()
    print('%s, %s' % (num, num / 2.0))
0, 0.0
1, 0.5
4, 2.0
9, 4.5
```

Как вам уже известно, так поступает каждый итерационный контекст: циклы `for`, встроенные функции `sum`, `map` и `sorted`, списковые включения и другие итерационные контексты, рассмотренные в главе 14, в том числе встроенные функции `any`, `all` и `list`. Будучи *итерируемыми объектами*, генераторные выражения могут появляться в любом итерационном контексте подобного рода в точности как результат вызова генераторной функции.

Например, в следующем коде генераторные выражения находятся в вызове строкового метода `join` и в присваивании кортежа, которые оба представляют собой итерационные контексты. В первом случае `join` запускает генератор и объединяет производимые им подстроки, ничего между ними не помещая — просто выполняет конкатенацию:

```
>>> ''.join(x.upper() for x in 'aaa,bbb,ccc'.split(','))
'AAABBBCCC'
>>> a, b, c = (x + '\n' for x in 'aaa,bbb,ccc'.split(','))
>>> a, c
('aaa\n', 'ccc\n')
```

Обратите внимание, что вызов `join` в предыдущем коде не требует *дополнительных* круглых скобок вокруг генератора. Согласно синтаксису, когда генераторное выражение является единственным элементом, который уже заключен в круглые скобки, предназначенные для других целей, то помещать его в еще одну пару круглых скобок *не требуется*. Однако во всех остальных случаях круглые скобки обязательны, даже если они кажутся излишними, как во втором вызове `sorted`:

```
>>> sum(x ** 2 for x in range(4))      # Круглые скобки необязательны
14
>>> sorted(x ** 2 for x in range(4))     # Круглые скобки необязательны
[0, 1, 4, 9]
>>> sorted((x ** 2 for x in range(4)), reverse=True) # Круглые скобки обязательны
[9, 4, 1, 0]
```

Подобно часто необязательным круглым скобкам в кортежах здесь нет широко принятого правила, хотя генераторное выражение не исполняет такой четкой роли, как фиксированная коллекция других объектов вроде кортежа, поэтому добавочные круглые скобки, вероятно, выглядят в нем паразитными.

Для чего используются генераторные выражения?

В точности как генераторные функции генераторные выражения обеспечивают оптимизацию *расхода памяти* — они не требуют создания сразу всего результирующего списка, что происходит в случае спискового включения в квадратных скобках. Также

подобно генераторным функциям они разделяют работу по выпуску результатов на небольшие *временные интервалы* – результаты выдаются постепенно вместо того, чтобы заставлять вызывающий код ожидать создания полного набора в единственном вызове.

С другой стороны, на практике генераторные выражения могут выполняться несколько медленнее списковых включений, а потому их лучше всего применять для очень крупных результирующих наборов или в приложениях, которые не могут ожидать генерации полных результатов. Но более авторитетное заявление о производительности придется отложить до написания сценариев для измерения времени выполнения в следующей главе.

Генераторные выражения тоже обладают преимуществами в плане *написания кода*, хотя и более субъективными, как показано в следующем разделе.

Генераторные выражения или `map`

Один из способов взглянуть на преимущества генераторных выражений в плане написания кода предусматривает их сравнение с другими инструментами для функционального программирования, как мы делали при исследовании списковых включений. Скажем, генераторные выражения часто эквивалентны вызовам `map` из Python 3.X, потому что в обоих случаях результирующие элементы генерируются по запросу. Тем не менее, аналогично списковым включениям генераторные выражения могут быть проще в написании, когда применяемой операцией оказывается не вызов функции. В Python 2.X функция `map` создает временные списки, а генераторные выражения – нет, но сравнение их кода все равно применимо:

```
>>> list(map(abs, (-1, -2, 3, 4)))          # Отображает функцию на кортеж
[1, 2, 3, 4]
>>> list(abs(x) for x in (-1, -2, 3, 4))      # Генераторное выражение
[1, 2, 3, 4]
>>> list(map(lambda x: x * 2, (1, 2, 3, 4)))    # Случай не функции
[2, 4, 6, 8]
>>> list(x * 2 for x in (1, 2, 3, 4))           # Проще как генератор?
[2, 4, 6, 8]
```

То же самое остается справедливым в сценариях использования для обработки текста вроде ранее показанного вызова `join` – списковое включение создает дополнительный временный список результатов, который в таком контексте будет совершенно бессмысленным, поскольку список не сохраняется, а `map` утрачивает преимущество простоты в сравнении с синтаксисом генераторных выражений, когда применяемая операция отличается от вызова:

```
>>> line = 'aaa,bbb,ccc'
>>> ''.join([x.upper() for x in line.split(',')])    # Создает бессмысленный
                                                       # список
'AAABBBCCC'
>>> ''.join(x.upper() for x in line.split(','))       # Генерирует результаты
'AAABBBCCC'
>>> ''.join(map(str.upper, line.split(',')))          # Генерирует результаты
'AAABBBCCC'
>>> ''.join(x * 2 for x in line.split(','))            # Проще как генератор?
'aaaaaaabbbbbbbcccccc'
>>> ''.join(map(lambda x: x * 2, line.split(',')))
'aaaaaaabbbbbbbcccccc'
```

Вызовы `map` и генераторные выражения могут также быть произвольно *вложенными*, что общеупотребительно в программах и требует вызова `list` или другого итерационного контекста для инициирования процесса выпуска результатов. Например, приведенное далее списковое включение производит такой же результат, как показанные после него эквиваленты в форме `map` из Python 3.X и генераторов, но создает два физических списка; остальные генерируют только по одному целому числу за раз с помощью вложенных генераторов, а форма генераторного выражения может более четко отражать свое намерение:

```
>>> [x * 2 for x in [abs(x) for x in (-1, -2, 3, 4)]] # Вложенные включения
[2, 4, 6, 8]
>>> list(map(lambda x: x * 2, map(abs, (-1, -2, 3, 4)))) # Вложенные отображения
[2, 4, 6, 8]
>>> list(x * 2 for x in (abs(x) for x in (-1, -2, 3, 4))) # Вложенные генераторы
[2, 4, 6, 8]
```

Хотя результатом всех трех форм является объединение операций, генераторы делают это, не создавая множество временных списков. В Python 3.X следующий пример вкладывает и комбинирует генераторы – вложенное генераторное выражение активируется посредством функции `map`, которая в свою очередь активируется только функцией `list`:

```
>>> import math
>>> list(map(math.sqrt, (x ** 2 for x in range(4)))) # Вложенные комбинации
[0.0, 1.0, 2.0, 3.0]
```

Говоря формально, функция `range` справа в Python 3.X тоже представляет собой генератор значений, активируемый самим генераторным выражением – *три уровня* генерации значений, которые производят индивидуальные значения от внутреннего уровня к внешнему только по запросу и которые “просто работают” благодаря инструментам и протоколу итерации Python. На самом деле генераторы можно произвольно смешивать и вкладывать, несмотря на то, что одни могут быть более допустимыми, чем другие:

```
>>> list(map(abs, map(abs, map(abs, (-1, 0, 1))))) # Вложение пришло в упадок?
[1, 0, 1]
>>> list(abs(x) for x in (abs(x) for x in (abs(x) for x in (-1, 0, 1))))
[1, 0, 1]
```

Последние примеры иллюстрируют, какими могут быть обычные генераторы, но они умышленно записаны в сложной форме, чтобы подчеркнуть тот факт, что генераторные выражения обладают аналогичным потенциалом злоупотребления, как и обсуждаемые ранее списковые включения – вы должны сохранять их простыми, если только они не обязаны быть сложными (позже в главе мы еще вернемся к данной теме).

Однако при правильном использовании генераторные выражения объединяют эффективность списковых включений с преимуществами, касающимися пространства и времени, других итерируемых объектов. Скажем, подходы без вложения предоставляют более простые решения, но по-прежнему задействуют сильные стороны генераторов – согласно девизу Python плоский в целом лучше, чем вложенный:

```
>>> list(abs(x) * 2 for x in (-1, -2, 3, 4)) # Эквиваленты без вложения
[2, 4, 6, 8]
>>> list(math.sqrt(x ** 2) for x in range(4)) # Плоский часто лучше
[0.0, 1.0, 2.0, 3.0]
>>> list(abs(x) for x in (-1, 0, 1))
[1, 0, 1]
```

Генераторные выражения или filter

Генераторные выражения также поддерживают весь обычный синтаксис списковых включений, в том числе конструкции if, которые работают подобно рассмотренному ранее вызову filter. Поскольку filter в Python 3.X является итерируемым объектом, который генерирует свои результаты по запросу, генераторное выражение с конструкцией if функционально эквивалентно (в Python 2.X функция filter производит временный список, чего не делает генератор, но снова сравнение их кода применимо). В следующем коде вызова join достаточно для принудительного выпуска результатов во всех формах:

```
>>> line = 'aa bbb c'  
>>> ''.join(x for x in line.split() if len(x) > 1)      # Генератор с  
                                                               # конструкцией if  
'aabbb'  
>>> ''.join(filter(lambda x: len(x) > 1, line.split()))  # Подобный вызов filter  
'aabbb'
```

Генератор выглядит минимально проще filter. Тем не менее, что касается списковых включений, то добавление шагов обработки к результатам filter также требует вызова map, который делает filter заметно сложнее генераторного выражения:

```
>>> ''.join(x.upper() for x in line.split() if len(x) > 1)  
'AABBB'  
>>> ''.join(map(str.upper, filter(lambda x: len(x) > 1, line.split())))  
'AABBB'
```

Фактически генераторные выражения делают для итерируемых объектов Python 3.X вроде создаваемых функциями map и filter то, что списковые включения делают для вариантов этих вызовов с построителями списков Python 2.X – они предоставляют более общие кодовые структуры, которые не опираются на функции, но по-прежнему откладывают выпуск результатов. Также подобно списковым включениям для генераторного выражения всегда имеется эквивалент на основе операторов, хотя временами он требует написания значительно большего объема кода:

```
>>> ''.join(x.upper() for x in line.split() if len(x) > 1)  
'AABBB'  
>>> res = ''  
>>> for x in line.split():      # Операторный эквивалент?  
    if len(x) > 1:            # Это также объединение  
        res += x.upper()  
  
>>> res  
'AABBB'
```

Однако в данном случае операторная форма не совсем такая же – она не может выпускать по одному элементу за раз и также эмулирует эффект метода join, который приводит к производству сразу всех результатов. Подлинным эквивалентом генераторного выражения была бы генераторная функция с оператором yield, как показано в следующем разделе.

Генераторные функции или генераторные выражения

Давайте подытожим то, что было рассмотрено до сих пор в разделе.

Генераторные функции

Определение `def`, содержащее оператор `yield`, превращается в генераторную функцию. При вызове она возвращает новый *объект генератора* с автоматическим сохранением локальной области видимости и местоположения в коде, автоматически созданным методом `__iter__`, который просто возвращает сам объект, и автоматически созданным методом `__next__` (`next` в Python 2.X), который запускает функцию или возобновляет ее выполнение с места, где она находилась в последний раз, и инициирует исключение `StopIteration`, когда выпуск результатов завершен.

Генераторные выражения

Выражение включения, помещенное в круглые скобки, известно как генераторное выражение. Оно возвращает новый *объект генератора* с таким же автоматически созданным интерфейсом в виде методов и сохранением состояния, как у генераторной функции, т.е. с методом `__iter__`, просто возвращающим сам объект, и методом `__next__` (`next` в Python 2.X), который запускает неявный цикл или возобновляет его выполнение с места, где он находился в последний раз, и инициирует исключение `StopIteration`, когда выпуск результатов завершен.

Совокупным эффектом является выпуск результатов по запросу в итерационных контекстах, которые задействуют такие интерфейсы автоматически.

Как вытекает из ряда предшествующих разделов, одну и ту же итерацию часто можно записать с помощью *либо* генераторной функции, *либо* генераторного выражения. Скажем, следующее генераторное выражение повторяет каждый символ в строке четыре раза:

```
>>> G = (c * 4 for c in 'SPAM')      # Генераторное выражение
>>> list(G)                      # Заставляет генератор выпустить все результаты
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

Эквивалентная генераторная функция требует чуть больше кода, но как функция с множеством операторов при необходимости она будет способна содержать больше логики и использовать больший объем информации о состоянии. По существу речь идет о таком же компромиссе, как между `lambda` и `def` в предыдущей главе — лаконичность выражения против моши оператора:

```
>>> def timesfour(S):          # Генераторная функция
    for c in S:
        yield c * 4

>>> G = timesfour('spam')
>>> list(G)                  # Автоматическая итерация
['ssss', 'pppp', 'aaaa', 'mmmm']
```

Для клиентов две версии больше похожи, нежели разнятся. Как выражения, так и функции поддерживают автоматическую и ручную итерацию — вызов `list` выше выполняет автоматическую итерацию, а в следующем коде итерация осуществляется вручную:

```
>>> G = (c * 4 for c in 'SPAM')
>>> I = iter(G)                # Итерация вручную (выражение)
>>> next(I)
'SSSS'
>>> next(I)
'PPPP'
```

```
>>> G = timesfour('spam')
>>> I = iter(G)                                # Итерация вручную (функция)
>>> next(I)
'ssss'
>>> next(I)
'pppp'
```

В обоих случаях Python автоматически создает объект генератора, который имеет оба метода, требуемые протоколом итерации, а также обеспечивает сохранение состояния для переменных из кода генератора и текущего местоположения в коде. Обратите внимание на то, что для повторной итерации мы создаем новые генераторы – как объясняется в следующем разделе, генераторы являются одноразовыми итераторами.

Тем не менее, для выражения, приведенного в конце предыдущего раздела, есть подлинный эквивалент на основе операторов: функция, которая выдает значения – хотя разница несущественна, если применяющий его код производит все результаты посредством инструмента, подобного `join`:

```
>>> line = 'aa bbb c'
>>> ''.join(x.upper() for x in line.split() if len(x) > 1)      # Выражение
'AABB'
>>> def gensub(line):                                              # Функция
    for x in line.split():
        if len(x) > 1:
            yield x.upper()
>>> ''.join(gensub(line))                                         # Но зачем генерировать?
'AABB'
```

Несмотря на то что генераторы способны исполнять адекватные роли, в случаях вроде представленного использование генераторов вместо показанного ранее простого операторного эквивалента может быть затруднительно оправдать кроме как стилистическими причинами. С другой стороны, обмен четырех строк кода на одну многим может показаться довольно убедительной стилистической причиной!

Генераторы являются объектами с одиночной итерацией

Тонкий, но важный момент: генераторные функции и генераторные выражения сами представляют собой итераторы и потому поддерживают только *одну активную итерацию* – в отличие от ряда встроенных типов нельзя иметь множество итераторов, каждый из которых находится в отличающейся позиции внутри набора результатов. Из-за этого итератором генератора является сам генератор; фактически, как указывалось ранее, вызов `iter` на генераторном выражении или функции необязателен и ничего не делает:

```
>>> G = (c * 4 for c in 'SPAM')
>>> iter(G) is G      # Итератором генератора является сам генератор:
                      #   G имеет метод __next__
True
```

В случае выполнения итерации по потоку результатов с помощью множества итераторов все они будут указывать на ту же самую позицию:

```
>>> G = (c * 4 for c in 'SPAM')      # Создать новый генератор
>>> I1 = iter(G)                      # Итерация вручную
>>> next(I1)
```

```
'SSSS'  
>>> next(I1)  
'PPPP'  
>>> I2 = iter(G)          # Второй итератор находится в той же самой позиции!  
>>> next(I2)  
'AAAA'
```

Более того, как только любая итерация доходит до завершения, все результаты оказываются израсходованными — чтобы начать сначала, нам придется создать новый генератор:

```
>>> list(I1)                # Собирает оставшиеся элементы I1  
['MMMM']  
>>> next(I2)                # Другие итераторы тоже израсходуются  
StopIteration  
>>> I3 = iter(G)            # То же самое касается новых итераторов  
>>> next(I3)  
StopIteration  
>>> I3 = iter(c * 4 for c in 'SPAM')    # Новый генератор, чтобы начать заново  
>>> next(I3)  
'SSSS'
```

То же самое остается справедливым для генераторных функций — приведенный ниже эквивалент на основе оператора `def` поддерживает только один активный итератор и израсходуется после одного прохода:

```
>>> def timesfour(S):  
    for c in S:  
        yield c * 4  
  
>>> G = timesfour('spam')      # Генераторные функции работают таким же образом  
>>> iter(G) is G  
True  
>>> I1, I2 = iter(G), iter(G)  
>>> next(I1)  
'ssss'  
>>> next(I1)  
'pppp'  
>>> next(I2)                  # I2 находится в той же позиции, что и I1  
'aaaa'
```

Такое поведение отличается от поведения ряда встроенных типов, которые поддерживают множество итераторов и проходов, а также отражают изменения на месте в активных итераторах:

```
>>> L = [1, 2, 3, 4]  
>>> I1, I2 = iter(L), iter(L)  
>>> next(I1)  
1  
>>> next(I1)  
2  
>>> next(I2)      # Списки поддерживают множество итераторов  
1  
>>> del L[2:]     # Изменения отражаются в итераторах  
>>> next(I1)  
StopIteration
```

Хотя указанная особенность не настолько очевидна в простых примерах, она может иметь значение в вашем коде: если вы хотите просматривать значения генератора много раз, тогда должны либо создавать новый генератор для каждого просмотра, либо построить из его значений список, допускающий многократные просмотры — значения единственного генератора будут потребляться и израсходоваться после одного прохода. Основной пример кода, который должен приспособливаться к такому свойству генераторов, предложен во врезке “Что потребует внимания: одноразовые итерации” далее в главе.

Начав писать код итерируемых объектов на основе классов в части VI второго тома, мы также увидим, что сами решаем, сколько итераций должны поддерживаться для объектов. В целом объекты, которым необходимо поддерживать множество проходов, будут возвращать вместо самих себя экземпляры добавочного класса. Предварительный обзор такой модели представлен в следующем разделе.

Расширение `yield from` из Python 3.3

В версии Python 3.3 появился расширенный синтаксис для оператора `yield`, который делает возможным делегирование работы подгенератору с помощью конструкции `from` генератор. В простых ситуациях он эквивалентен выдаче в цикле `for` — в следующем коде вызов `list` заставляет генератор выпустить все свои значения, а включение в круглых скобках является генераторным выражением, раскрытым в данной главе:

```
>>> def both(N):
    for i in range(N): yield i
    for i in (x ** 2 for x in range(N)): yield i

>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]
```

Новый синтаксис Python 3.3 возможно делает это более лаконичным и явным и поддерживает все обычные контексты применения генераторов:

```
>>> def both(N):
    yield from range(N)
    yield from (x ** 2 for x in range(N))

>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]

>>> ' : '.join(str(i) for i in both(5))
'0 : 1 : 2 : 3 : 4 : 0 : 1 : 4 : 9 : 16'
```

Однако в более развитых ролях такое расширение позволяет подгенераторам получать значения, переданные методами `send` и `throw`, напрямую из вызывающей области видимости и возвращать финальное значение внешнему генератору. В итоге появляется возможность разделения таких генераторов на множество подгенераторов, очень похожую на возможность разделения одиночной функции на несколько подфункций.

Поскольку расширение `yield from` доступно, только начиная с Python 3.3, и выходит за рамки общего описания генераторов, дополнительные сведения ищите в руководствах по языку. Еще один пример использования `yield from` можно найти в решении упражнения 11, приведенного в конце главы 21.

Генерация во встроенных типах, инструментах и классах

Наконец, хотя основное внимание в текущем разделе сосредоточено на написании самих генераторов значений, не следует забывать о том, что многие встроенные типы ведут себя похожим образом — например, как было показано в главе 14, *словари* являются итерируемыми объектами с итераторами, которые выдают ключи на каждой итерации:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> x = iter(D)
>>> next(x)
'c'
>>> next(x)
'b'
```

Подобно значениям, производимым самостоятельно написанными генераторами, по ключам словаря можно проходить вручную и с применением автоматических итерационных средств, в том числе циклов `for`, вызовов `map`, списковых включений и многих других контекстов, встречавшихся в главе 14:

```
>>> for key in D:
    print(key, D[key])

c 3
b 2
a 1
```

Как уже известно, для *файловых* итераторов Python просто загружает строки из файла по запросу:

```
>>> for line in open('temp.txt'):
    print(line, end='')

Tis but
a flesh wound.
```

В то время как итерируемые объекты встроенных типов привязаны к специальному виду генерации значений, концепция подобна многоцелевым генераторам, которые мы пишем с использованием выражений и функций. Итерационные контексты вроде циклов `for` принимают любой итерируемый объект, который имеет ожидаемые методы, будь он определяемым пользователем или встроенным.

Генераторы и библиотечные средства: инструменты прохода по каталогам

Хотя это выходит за рамки тематики книги, следует отметить, что в наши дни многие стандартные библиотечные средства Python тоже генерируют значения, включая анализаторы сообщений электронной почты и стандартный *инструмент прохода по каталогам*, который на каждом уровне дерева каталогов выдает кортеж с текущим каталогом, его подкаталогами и имеющимися файлами:

```
>>> import os
>>> for (root, subs, files) in os.walk('.'):
    # Генератор инструмента
    # прохода по каталогам
    for name in files:
        # Операция 'поиска' в Python
        if name.startswith('call'):
            print(root, name)

. callables.py
.\dualpkg callables.py
```

На самом деле `os.walk` в Python реализована как рекурсивная функция в стандартном библиотечном файле `os.py`, находящемся в `C:\Python37\Lib` на компьютере Windows. Поскольку для возвращения результатов в ней применяется `yield` (и `yield from` вместо `for`, начиная с версии Python 3.3), она является нормальной генераторной функцией и потому итерируемым объектом:

```
>>> G = os.walk(r'C:\code\pkg')
>>> iter(G) is G      # Однопроходный итератор: вызов iter(G) необязателен
True
>>> I = iter(G)
>>> next(I)
('C:\\code\\pkg', ['__pycache__'], ['eggs.py', 'eggs.pyc', 'main.py', ... и
так далее...])
>>> next(I)
('C:\\code\\pkg\\__pycache__', [], ['eggs.cpython-37.pyc', ... и так далее...])
>>> next(I)
StopIteration
```

За счет выдачи результатов по мере продвижения инструмент прохода по каталогам не требует от своих клиентов ждать, пока не будет пройдено все дерево. Дополнительные сведения о данном инструменте ищите в руководствах по Python и в книге *Programming Python* (<http://www.oreilly.com/catalog/9780596158101>). Кроме того, в главе 14 и других главах упоминается `os.popen` – связанный итерируемый объект, используемый для запуска команды оболочки и чтения ее вывода.

Генераторы и применение функций

В главе 18 отмечалось, что аргументы со звездочкой способны распаковывать *итерируемый объект* в индивидуальные аргументы. Ознакомившись с генераторами, мы можем также посмотреть, что это означает в коде. Вот пример для Python 3.X и 2.X (хотя `range` в Python 2.X дает список):

```
>>> def f(a, b, c): print('%s, %s, and %s' % (a, b, c))
>>> f(0, 1, 2)                      # Нормальные позиционные аргументы
0, 1, and 2
>>> f(*range(3))                  # Распаковка значений range:
                                         # итерируемый объект в Python 3.X
0, 1, and 2
>>> f(*(i for i in range(3)))    # Распаковка значений генераторного выражения
0, 1, and 2
```

Прием применим также к словарям и представлениям (хотя `dict.values` – список в Python 2.X, а при передаче значений по ключам порядок произведен):

```
>>> D = dict(a='Bob', b='dev', c=40.5); D
{'b': 'dev', 'c': 40.5, 'a': 'Bob'}
>>> f(a='Bob', b='dev', c=40.5)    # Нормальные ключевые аргументы
Bob, dev, and 40.5
>>> f(**D)                        # Распаковка словаря: ключ=значение
Bob, dev, and 40.5
>>> f(*D)                          # Распаковка итератора ключей
b, c, and a
>>> f(*D.values())                # Распаковка итератора представления:
                                         # итерируемый объект в Python 3.X
dev, 40.5, and Bob
```

Поскольку встроенная функция print в Python 3.X выводит все переменное количество своих аргументов, следующие далее формы будут эквивалентными – в последней используется * для распаковки результатов, поступивших из генераторного выражения (во второй форме также создается список возвращаемых значений, а первая может оставить курсор в конце строки вывода в некоторых оболочках, но не в графическом пользовательском интерфейсе IDLE):

```
>>> for x in 'spam': print(x.upper(), end=' ')
S P A M
>>> list(print(x.upper(), end=' ') for x in 'spam')
S P A M [None, None, None, None]
>>> print(*(x.upper() for x in 'spam'))
S P A M
```

Дополнительный пример распаковки строк, переданных файловым итератором, в аргументы приводился в главе 14.

Предварительный обзор: итерируемые объекты, определяемые пользователем в классах

Несмотря на что данная тема выходит за рамки настоящей главы, следует упомянуть о возможности реализации произвольных определяемых пользователем объектов генераторов с помощью *классов*, которые адаптированы к протоколу итерации. В таких классах определен специальный метод __iter__, запускаемый встроенной функцией iter. Функция iter в свою очередь возвращает объект, который имеет метод __next__ (next в Python 2.X), запускаемый встроенной функцией next:

```
class SomeIterable:
    def __iter__(...): ...      # Запускается iter(): возвращает текущий или
                                # дополнительный объект
    def __next__(...): ...       # Запускается next(): реализуется здесь или
                                # в другом классе
```

Как упоминалось в предыдущем разделе, эти классы обычно возвращают свои объекты напрямую для поведения с единственной итерацией или дополнительный объект для поддержки множества проходов.

В качестве альтернативы функции методов в определяемом пользователем классе итерируемых объектов временами может использоваться yield для превращения их самих в генераторы с автоматически создаваемым методом __next__ – распространенное применение yield, демонстрируемое в главе 30, которое совершенно неявное и потенциально полезное! Метод индексации __getitem__ также доступен в виде запасного варианта для итерации, хотя он часто не настолько гибкий, как схема с __iter__ и __next__ (но обладает преимуществами при работе с последовательностями).

Объекты экземпляров, созданные из такого класса, считаются итерируемыми и могут использоваться в циклах for и всех остальных итерационных контекстах. Тем не менее, посредством классов мы имеем доступ к развитым возможностям логики и структурирования данных, таким как наследование, которое другие генераторные конструкции сами по себе предложить не в состоянии. За счет написания кода методов классы также могут делать поведение итерации более явным, нежели “магические” объекты итераторов, ассоциированные со встроенными типами и генераторными функциями и выражениями (правда, классы обладают собственной магией).

Таким образом, история с итераторами и генераторами фактически не может быть завершена до тех пор, пока мы не увидим, как она соотносится с классами. В текущий момент мы должны довольствоваться тем, что отложим ее окончание — и финальный результат — до обсуждения итерируемых объектов, основанных на классах, в главе 30.

Пример: генерация перемешанных последовательностей

Чтобы продемонстрировать мощь итерационных инструментов в действии, давайте обратимся к более полным сценариям применения. В главе 18 была написана тестовая функция, которая перемешивала порядок следования аргументов, используемых для проверки работоспособности обобщенных функций пересечения и объединения. Там отмечалось, что возможно было бы лучше ее реализовать в виде генератора значений. Теперь, когда известно, как писать генераторы, это послужит иллюстрацией практического применения.

Одно благородное замечание: поскольку все примеры в этом разделе (включая перестановки в конце) выполняют срез и конкатенацию объектов, они работают только с последовательностями вроде строк и списков, но не с произвольными *итерируемыми объектами* наподобие файлов, отображений и других генераторов. То есть некоторые примеры сами по себе будут генераторами, производя значения по запросу, но они не могут обрабатывать генераторы как входные данные. Обобщение для более широких категорий остается открытой задачей, хотя код здесь в основном не придется изменять, если перед передачей генераторов, отличных от последовательностей, помещать их внутрь вызовов `list`.

Перемешивание последовательностей

Как было показано в главе 18, мы можем переупорядочивать последовательность посредством нарезания и конкатенации, перемещая головной элемент в конец на каждой итерации цикла; использование *нарезания* вместо индексации элемента позволяет операции `+` работать для произвольных типов последовательностей:

```
>>> L, S = [1, 2, 3], 'spam'
>>> for i in range(len(S)):
...     # Цикл с подсчетом 0..3
...     S = S[1:] + S[:1]          # Перемещение головного элемента в конец
...     print(S, end=' ')
pams amsp mspa spam
>>> for i in range(len(L)):
...     L = L[1:] + L[:1]          # Нарезание, поэтому работает любой тип
...                               # последовательности
...     print(L, end=' ')
[2, 3, 1] [3, 1, 2] [1, 2, 3]
```

Как объяснялось в главе 13, добиться тех же результатов по-другому можно путем перемещения целого головного раздела в конец, но порядок результатов слегка варьируется:

```
>>> for i in range(len(S)):    # Для позиций 0..3
...     X = S[i:] + S[:i]      # Задняя часть + передняя часть (тот же самый эффект)
...     print(X, end=' ')
spam pams amsp mspa
```

Простые функции

В том виде, как есть, написанный код работает только с особым образом именованными переменными. С целью его обобщения мы можем превратить этот код в *простую функцию*, которая будет работать с любым объектом, переданным в аргументе, и возвращать результат. Так как в первой версии явно просматривалась классическая схема со списковым включением, мы проделаем определенную работу, чтобы сохранить ее во второй версии:

```
>>> def scramble(seq):
    res = []
    for i in range(len(seq)):
        res.append(seq[i:] + seq[:i])
    return res

>>> scramble('spam')
['spam', 'pams', 'amsp', 'mspa']

>>> def scramble(seq):
    return [seq[i:] + seq[:i] for i in range(len(seq))]

>>> scramble('spam')
['spam', 'pams', 'amsp', 'mspa']

>>> for x in scramble((1, 2, 3)):
    print(x, end=' ')
(1, 2, 3) (2, 3, 1) (3, 1, 2)
```

Здесь можно было бы также использовать рекурсию, но в таком контексте она видимо оказалась бы излишней.

Генераторные функции

Простой подход в предыдущем разделе работает, но он требует построения в памяти всего результирующего списка (что не очень хорошо в плане расхода памяти, если список крупный) и заставляет вызывающий код ожидать до тех пор, пока полный список не будет готов (что также не идеально, когда построение занимает значительное время). Мы можем улучшить положение дел, превратив код в *генераторную функцию*, которая выдает по одному результату за раз, с применением одной из двух схем написания кода:

```
>>> def scramble(seq):
    for i in range(len(seq)):
        seq = seq[1:] + seq[:1]      # Генераторная функция
        yield seq                   # Здесь работают присваивания

>>> def scramble(seq):
    for i in range(len(seq)):
        yield seq[i:] + seq[:i]    # Генераторная функция
                                    # Выдача по одному элементу на итерацию

>>> list(scramble('spam'))      # list() генерирует все результаты
['spam', 'pams', 'amsp', 'mspa']
>>> list(scramble((1, 2, 3)))   # Подходит любой тип последовательности
[(1, 2, 3), (2, 3, 1), (3, 1, 2)]
>>>
>>> for x in scramble((1, 2, 3)): # Цикл for генерирует результаты
    print(x, end=' ')
(1, 2, 3) (2, 3, 1) (3, 1, 2)
```

Генераторные функции сохраняют состояние своих локальных областей видимости, пока активны, минимизируют требования относительно пространства памяти и разделяют работу на более короткие временные интервалы. Будучи полноценными функциями, они также весьма универсальны. Важно отметить, что циклы `for` и другие итерационные инструменты работают одинаково при проходе по реально существующему списку и генератору значений — функция может свободно выбирать одну из двух схем и даже изменять стратегию в будущем.

Генераторные выражения

Ранее уже было показано, что *генераторные выражения* (включения в круглых, а не в квадратных скобках) также генерируют значения по запросу и сохраняют свое локальное состояние. Они не настолько гибкие, как полные функции, но из-за автоматической выдачи своих значений выражения часто могут оказываться лаконичнее в специфических сценариях использования наподобие следующего:

```
>>> S
'spam'
>>> G = (S[i:] + S[:i] for i in range(len(S)))      # Эквивалентное
                                                # генераторное выражение
>>> list(G)
['spam', 'pams', 'amsp', 'mspa']
```

Обратите внимание, что мы не можем здесь применять оператор присваивания из первой версии генераторной функции, т.к. операторы в генераторных выражениях не разрешены. В итоге их область действия несколько сужается, но во многих ситуациях выражения способны выполнять похожую работу, как иллюстрируется в настоящем разделе. Чтобы обобщить генераторное выражение для произвольного объекта, его понадобится поместить внутрь простой функции, принимающей аргумент и возвращающей генератор, который его использует:

```
>>> F = lambda seq: (seq[i:] + seq[:i] for i in range(len(seq)))
>>> F(S)
<generator object <genexpr> at 0x0000000029883F0>
>>>
>>> list(F(S))
['spam', 'pams', 'amsp', 'mspa']
>>> list(F([1, 2, 3]))
[[1, 2, 3], [2, 3, 1], [3, 1, 2]]
>>> for x in F((1, 2, 3)):
...     print(x, end=' ')
...
(1, 2, 3) (2, 3, 1) (3, 1, 2)
```

Клиент для тестирования

Наконец, мы можем применять генераторную функцию или ее эквивалент в виде выражения в функции `tester` из главы 18 для производства перетасованных аргументов — функция перемешивания последовательностей становится инструментом, который допускается использовать в других контекстах:

```
# Файл scramble.py
def scramble(seq):
    for i in range(len(seq)):      # Генераторная функция
        yield seq[i:] + seq[:i]    # Выдача по одному элементу на итерацию
scramble2 = lambda seq: (seq[i:] + seq[:i] for i in range(len(seq)))
```

За счет выноса генерации значений во внешний инструмент функция `tester` становится проще:

```
>>> from scramble import scramble
>>> from inter2 import intersect, union
>>>
>>> def tester(func, items, trace=True):
    for args in scramble(items):      # Использовать генератор
                                         # (или scramble2(items))
        if trace: print(args)
        print(sorted(func(*args)))
>>> tester(intersect, ('aab', 'abcde', 'ababab'))
('aab', 'abcde', 'ababab')
['a', 'b']
('abcde', 'ababab', 'aab')
['a', 'b']
('ababab', 'aab', 'abcde')
['a', 'b']
>>> tester(intersect, ([1, 2], [2, 3, 4], [1, 6, 2, 7, 3]), False)
[2]
[2]
[2]
```

Перестановки: все возможные комбинации

С описанными методиками связаны многие другие применения в реальных приложениях – взять хотя бы генерацию вложений в сообщении электронной почты или точек, подлежащих отображению в графическом пользовательском интерфейсе. Более того, другие виды перемешивания последовательностей играют центральную роль в других приложениях, начиная с поисковых и заканчивая математическими. В том виде как есть, наша функция перемешивания последовательностей выполняет простое переупорядочивание, но определенным программам нужен более полный набор всех возможных порядков, который мы получаем из *перестановок*, производимых с использованием рекурсивных функций в формах построителя списка и генератора из следующего файла модуля:

```
# Файл permute.py
def permute1(seq):
    if not seq:                      # Тасование любой последовательности: список
        return [seq]                  # Пустая последовательность
    else:
        res = []
        for i in range(len(seq)):
            rest = seq[:i] + seq[i+1:]  # Удаление текущего узла
            for x in permute1(rest):   # Перестановка остальных
                res.append(seq[i:i+1] + x) # Добавление узла спереди
        return res
def permute2(seq):
    if not seq:                      # Тасование любой последовательности: генератор
        yield seq                   # Пустая последовательность
    else:
        for i in range(len(seq)):
            rest = seq[:i] + seq[i+1:]  # Удаление текущего узла
            for x in permute2(rest):   # Перестановка остальных
                yield seq[i:i+1] + x    # Добавление узла спереди
```

Обе функции производят одинаковые результаты, хотя вторая откладывает выполнение большей части своей работы до тех пор, пока у нее не будет запрошен результат. Код чуть сложнее, особенно во второй функции (и некоторым новичкам в Python он может даже показаться жестоким и бесчеловечным наказанием!). Однако, как вскоре выяснится, есть сценарии, где подход с генератором может быть крайне удобным.

Изучите и протестируйте приведенный код, чтобы лучше его понять, и добавьте вывод для его трассировки, если это поможет. Если он по-прежнему выглядит загадочным, попробуйте разобраться сначала с первой версией; вспомните, что генераторные функции просто возвращают объекты посредством методов, которые поддерживают операции `next`, выполняемые циклами `for` на каждом уровне, и не производят результатов до тех пор, пока не будут подвергнуты итерации. Кроме того, отслеживайте некоторые из представляемых далее примеров, чтобы посмотреть, как с ними справляется данный код.

Перестановки производят больше порядков, чем первоначальная функция тасования – для N элементов мы получаем $N!$ (факториал) результатов, а не просто N (24 результата для 4 элементов: $4 * 3 * 2 * 1$). На самом деле именно поэтому здесь необходима *рекурсия*: количество вложенных циклов произвольно и зависит от длины представляющей последовательности:

```
>>> from scramble import scramble
>>> from permute import permute1, permute2
>>> list(scramble('abc'))                                # Простое перемешивание: N
['abc', 'bca', 'cab']
>>> permute1('abc')                                     # Перестановок больше: N!
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> list(permute2('abc'))                               # Генерация всех комбинаций
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> G = permute2('abc')                                 # Итерация (iter() не требуется)
>>> next(G)
'abc'
>>> next(G)
'acb'
>>> for x in permute2('abc'): print(x)                 # Автоматическая итерация
...ВЫВОДИТ ШЕСТЬ СТРОК...
```

Результаты версий со списком и генератором одинаковы, хотя генератор минимизирует потребление памяти и время ожидания результатов. Для большего числа элементов набор всех перестановок будет намного большим, чем дает простое перемешивание:

```
>>> permute1('spam') == list(permute2('spam'))
True
>>> len(list(permute2('spam'))), len(list(scramble('spam')))
(24, 4)
>>> list(scramble('spam'))
['spam', 'pams', 'amsp', 'mspa']
>>> list(permute2('spam'))
['spam', 'spma', 'sapm', 'samp', 'smpa', 'smap', 'psam', 'psma', 'pasm',
 'pams',
 'pmsa', 'pmas', 'aspm', 'asmp', 'apsm', 'apms', 'amsp', 'amps', 'mspa',
 'msap',
 'mps', 'mpsa', 'masp', 'maps']
```

Согласно главе 19 существуют также нерекурсивные альтернативы, применяющие явные стеки и очереди, к тому же распространены другие порядки последовательностей (например, поднаборы фиксированного размера и комбинации, которые отфильтровывают дубликаты в другом порядке), но они требуют использования расширений, от чего мы воздержимся. Дополнительные сведения по этой теме ищите в книге *Programming Python* (<http://www.oreilly.com/catalog/9780596158101>) или поэкспериментируйте самостоятельно.

Не злоупотребляйте генераторами: EIBTI

Генераторы — довольно сложный инструмент, который возможно лучше трактовать как необязательную тему, если не учитывать тот факт, что они буквально пронизывают язык Python, особенно в линейке Python 3.X. На самом деле для аудитории этой книги генераторы представляются даже не такими необязательными, как тема Unicode (которая вынесена в часть VIII). Как выяснилось, фундаментальные встроенные инструменты, подобные `range`, `map`, `keys` в словарях и файлам, теперь являются генераторами, так что вы должны быть знакомы с концепцией, даже если самостоятельно не пишете новые генераторы. Кроме того, определяемые пользователем генераторы становятся все более распространенными в коде Python, который вы можете встречать в нынешнее время — скажем, в стандартной библиотеке Python.

В целом те же самые предостережения, которые давались в отношении списковых включений, применимы и здесь: не усложняйте свой код определяемыми пользователем генераторами, когда это неоправданно. Веские причины использовать такие инструменты могут отсутствовать, особенно для небольших программ и наборов данных. В подобных ситуациях достаточно будет простых списков результатов, которые легче для понимания, автоматически подвергаются сборке мусора и могут выпускаться быстрее (что так и есть в наши дни: см. следующую главу). С такими расширенными инструментами, как генераторы, полагающимся на неявную “магию”, может быть интересно экспериментировать, но им не место в реальном коде, который должен применяться другими, кроме случаев, когда это явно оправданно.

Или снова прибегнем к девизам, выводимым `import this`:

Явное лучше неявного.

Аббревиатура *EIBTI* англоязычной формулировки девиза “*explicit is better than implicit*” является одним из основных руководящих принципов Python и на то имеется веская причина: чем более явно код сообщает о своем поведении, тем более вероятно, что следующий программист сумеет его понять. Сказанное напрямую касается генераторов, чье неявное поведение вполне может оказаться трудным для понимания, нежели более ясные альтернативы. Всегда сохраняйте код простым, если только он не обязан быть сложным!

Обратная сторона: пространство памяти и время, лаконичность, выразительность

Следует отметить, что существуют специфические сценарии использования, для которых генераторы способны предложить хорошие решения. Они могут сократить объем памяти, занимаемой некоторыми программами, уменьшить задержки в других программах и временами делать невозможное возможным. Например, возьмем программу, которая должна производить все возможные перестановки нетривиальной последовательности. Поскольку количество перестановок равно *факториалу*, растущему экспоненциально, представленная ранее рекурсивная функция `permute1` с постро-

ением списка либо введет заметную и возможно бесконечную паузу, либо потерпит неудачу из-за высоких требований к памяти. С другой стороны, рекурсивный генератор `permute2` справляется с задачей, т.к. быстро возвращает индивидуальные результаты и может обрабатывать очень крупные наборы значений:

```
>>> import math
>>> math.factorial(10)      # 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
3628800
>>> from permute import permute1, permute2
>>> seq = list(range(10))
>>> p1 = permute1(seq)      # 37 секунд на компьютере с четырехядерным
                           # процессором 2 ГГц
                           # Создает список из более 3,6 миллиона чисел
>>> len(p1), p1[0], p1[1]
(3628800, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7, 9, 8])
```

В данном случае функция `permute1` сделала на моем компьютере паузу длительностью 37 секунд, пока строила список из более 3,6 миллиона элементов, но функция `permute2` может начать возвращение результатов незамедлительно:

```
>>> p2 = permute2(seq)      # Генератор немедленно возвращает управление
>>> next(p2)               # И быстро производит каждый результат по запросу
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> next(p2)
[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]

>>> p2 = list(permute2(seq))    # Около 28 секунд, хотя в этом нет смысла
>>> p1 == p2                  # Генерируется точно такой же набор результатов
True
```

Естественно, мы могли бы оптимизировать код функции построения списка, чтобы он выполнялся быстрее (скажем, применение явного стека вместо рекурсии изменило бы ее производительность), но для более крупных последовательностей эта функция вообще неприемлема — для всего лишь 50 элементов количество перестановок не позволяет построить список результатов, требуя ожидания слишком долгого времени для простых смертных вроде нас (и более высокие значения превысят заранее установленный предел глубины стека рекурсии: см. предыдущую главу). Тем не менее, генератор вполне жизнеспособен — он в состоянии производить индивидуальные результаты незамедлительно:

```
>>> math.factorial(50)
304140932017133780436126081660647688443776415689605120000000000000
>>> p3 = permute2(list(range(50)))
>>> next(p3)                 # permute1 здесь не подходит!
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49]
```

Ради еще большей забавы, а также для выдачи более изменчивых и менее детерминированных результатов, мы могли бы также использовать модуль Python по имени `random` из главы 5, чтобы случайным образом тасовать последовательность до того, как функция перестановок начнет свою работу. (Фактически в общем случае допускается применять функцию случайного тасования в качестве генератора перестановок, когда мы либо можем предположить, что она не будет повторять тасования, пока мы их потребляем, либо проверять ее результаты перед тасованием во избежание повторений — и уповать на то, что мы не живем в странной вселенной, где случайная последовательность повторяет тот же самый результат бесконечное количество раз!)

В следующем примере каждый вызов `permute2` и `next` возвращает управление немедленно, как и ранее, но вызов `permute1` зависает:

```
>>> import random
>>> math.factorial(20)      # permute1 здесь не подойдет
2432902008176640000
>>> seq = list(range(20))
>>> random.shuffle(seq)    # Предварительно перетасовать последовательность
                           # случайным образом
>>> p = permute2(seq)
>>> next(p)
[10, 17, 4, 14, 11, 3, 16, 19, 12, 8, 6, 5, 2, 15, 18, 7, 1, 0, 13, 9]
>>> next(p)
[10, 17, 4, 14, 11, 3, 16, 19, 12, 8, 6, 5, 2, 15, 18, 7, 1, 0, 9, 13]
>>> random.shuffle(seq)
>>> p = permute2(seq)
>>> next(p)
[16, 1, 5, 14, 15, 12, 0, 2, 6, 19, 10, 17, 11, 18, 13, 7, 4, 9, 8, 3]
>>> next(p)
[16, 1, 5, 14, 15, 12, 0, 2, 6, 19, 10, 17, 11, 18, 13, 7, 4, 9, 3, 8]
```

Суть здесь в том, что генераторы иногда способны производить результаты из крупных наборов решений, тогда как построители списков – нет. В то же время неясно, насколько распространены такие сценарии использования в реальном мире, и это не обязательно служит оправданием *неявной* природы генерации значений, которую привносят генераторные функции и выражения. Как будет показано в части VI, генерацию значений можно также реализовать в виде итерируемых объектов, создаваемых посредством *классов*. Итерируемые объекты, основанные на классах, тоже могут выпускать элементы по запросу, к тому же они гораздо более явные, чем магические объекты и методы, производимые для генераторных функций и выражений.

Одной из задач программирования является нахождение баланса между подобного рода компромиссами, и каких-то абсолютных правил не существует. Хотя преимущества генераторов могут временами оправдывать их применение, удобство сопровождения всегда должно иметь высший приоритет. Подобно включениям генераторы также предлагают *выразительность и экономию кода*, перед чем трудно устоять, если вы понимаете, как генераторы работают, но это нужно соотнести с возможным разочарованием коллег, имеющих иное мнение.

Пример: эмуляция `zip` и `map` с помощью итерационных инструментов

Чтобы помочь в дальнейшей оценке ролей генераторов, давайте рассмотрим еще один пример их в действии, который продемонстрирует, насколько они могут быть выразительными. Как только вы узнаете о включениях, генераторах и других итерационных инструментах, сразу выяснится, что эмуляция работы многих встроенных функций Python оказывается прямолинейной и поучительной. Скажем, мы уже видели, что встроенные функции `zip` и `map` соответственно объединяют итерируемые объекты и проецируют на них функции. При множестве итерируемых аргументов `map` проецирует функцию на элементы, взятые из каждого итерируемого объекта, во многом таким же способом, которым `zip` объединяет их в пары (функция `map` в Python 3.X усекает до более короткого итерируемого объекта, а в Python 2.X дополняет более короткие итерируемые объекты значением `None`):

```

>>> S1 = 'абв'
>>> S2 = 'хyz123'
>>> list(zip(S1, S2))                                # zip объединяет в пары элементы
                                                       # из итерируемых объектов
[('а', 'х'), ('б', 'у'), ('в', 'з')]
# zip объединяет элементы в пары, усекает до самого короткого
>>> list(zip([-2, -1, 0, 1, 2]))                  # Единственная последовательность:
                                                       # 1-арные кортежи
[(-2,), (-1,), (0,), (1,), (2,)]
>>> list(zip([1, 2, 3], [2, 3, 4, 5]))            # N последовательностей:
                                                       # N-арные кортежи
[(1, 2), (2, 3), (3, 4)]
# map передает объединенные в пары элементы функции, усекает
>>> list(map(abs, [-2, -1, 0, 1, 2]))             # Единственная последовательность:
                                                       # 1-арная функция
[2, 1, 0, 1, 2]
>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5]))       # N последовательностей:
                                                       # N-арная функция, Python 3.X
[1, 8, 81]

# map и zip принимают произвольные итерируемые объекты
>>> list(map(lambda x, y: x + y, open('script2.py'), open('script2.py')))
['import sys\nimport sys\n', 'print(sys.path)\nprint(sys.path)\n', ...и так далее...]
>>> [x + y for (x, y) in zip(open('script2.py'), open('script2.py'))]
['import sys\nimport sys\n', 'print(sys.path)\nprint(sys.path)\n', ...и так далее...]

```

Несмотря на использование для разных целей, если вы достаточно хорошо изучите приводимые примеры, то сможете заметить отношение между результатами `zip` и аргументами функции `map`, которое задействовано в следующем примере.

Написание собственной функции `map(func, ...)`

Хотя встроенные функции `map` и `zip` отличаются высокой скоростью и удобством, их всегда возможно эмулировать в собственном коде. Например, в предыдущей главе мы видели функцию, которая эмулировала встроенную функцию `map` для единственного аргумента в форме последовательности (или другого итерируемого объекта). Она требовала гораздо большей работы, чтобы разрешить передачу множества последовательностей, как делает встроенная функция:

```

# Аналог map(func, seqs...) на основе zip
def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))

```

Версия `mymap` в значительной степени опирается на специальный синтаксис передачи аргументов `*аргументы` — она накапливает множество аргументов в виде последовательностей (в действительности итерируемых объектов), распаковывает их как аргументы `zip` для объединения и затем распаковывает результирующие пары `zip` как аргументы для переданной функции. То есть мы задействуем тот факт, что объединение в пары по существу представляет собой вложенную операцию в отображении.

Тестовый код в конце применяет это к одной и двум последовательностям для производства показанного далее вывода — того же самого, который мы получили бы с помощью встроенной функции `map` (если хотите самостоятельно запустить код, то он находится в файле `тумар.py` внутри пакета примеров книги):

```
[2, 1, 0, 1, 2]
[1, 8, 81]
```

Правда, в предыдущей версии несложно заметить классическую *схему спискового включения*, строящую список результатов операции в цикле `for`. Мы можем записать код функции `тумар` более компактно как эквивалентное односстрочное списковое включение:

```
# Использование спискового включения
def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]
print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

Результаты запуска оказываются такими же, как ранее, но код лаконичнее и может выполнятся быстрее (производительность более подробно обсуждается в разделе “Измерение времени выполнения итерационных альтернатив” главы 21). Однако обе написанные выше версии `тумар` строят результирующие списки целиком, что может привести к излишнему расходу памяти в случае крупных списков. Теперь, когда нам известны *генераторные функции и выражения*, довольно легко переписать обе альтернативные версии для производства результатов по запросу:

```
# Использование генераторов: yield и ...
def mymap(func, *seqs):
    for args in zip(*seqs):
        yield func(*args)
def mymap(func, *seqs):
    return (func(*args) for args in zip(*seqs))
```

Новые версии выпускают такие же результаты, но возвращают генераторы, предназначенные для поддержки протокола итерации — первая версия выдает по одному результату за раз, а вторая возвращает результат генераторного выражения, чтобы делать то же самое. Они дадут такие же результаты, если их поместить внутрь вызовов `list` для принудительного производства сразу всех значений:

```
print(list(mymap(abs, [-2, -1, 0, 1, 2])))
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))
```

В действительности здесь никакая работа не делается до тех пор, пока вызовы `list` не заставят генераторы выполнятся, активируя протокол итерации. Генераторы, возвращаемые самими нашими функциями, а также версией `zip` из Python 3.X, которую они используют, производят результаты только по запросу.

Написание собственных функций `zip(...)` и `map(None, ...)`

Разумеется, порядочная часть магии в показанных до сих пор примерах была связана с применением встроенной функции `zip` для объединения в пары элементов из множества последовательностей или итерируемых объектов, переданных в аргументах. Наши аналоги `map` также действительно эмулировали поведение версии `map` из Python 3.X — они выполняли усечение по длине самой короткой последовательности и

не поддерживали понятие дополнения результатов, когда длины последовательностей отличаются, как делает версия `map` в Python 2.X с аргументом `None`:

```
C:\code> c:\python27\python
>>> map(None, [1, 2, 3], [2, 3, 4, 5])
[(1, 2), (2, 3), (3, 4), (None, 5)]
>>> map(None, 'abc', 'xyz123')
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

Используя итерационные инструменты, мы можем написать код аналогов, которые эмулируют и усечение `zip`, и дополнение `map` из Python 2.X – как оказалось, их код почти одинаков:

```
# Аналоги zip(seqs...) и map(None, seqs...) из Python 2.X
def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    res = []
    while any(seqs):
        res.append(tuple((S.pop(0) if S else pad) for S in seqs))
    return res

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))
```

Обе написанные функции работают с *итерируемым* объектом любого типа, поскольку они прогоняют свои аргументы через встроенную функцию `list`, чтобы инициировать генерацию результатов (например, вдобавок к последовательностям наподобие строк в качестве аргументов подошли бы файлы). Обратите внимание на применение встроенных функций `all` и `any` – они возвращают `True`, если соответственно все или любые элементы в итерируемом объекте являются `True` (или непустыми, что эквивалентно). Указанные встроенные функции используются для останова цикла, когда любой или все перечисленные аргументы становятся пустыми после удалений.

Также следует отметить применение аргумента `pad` с *передачей только по ключевому слову* из Python 3.X. В отличие от функции `map` в Python 2.X наша версия позволит указывать любой дополняющий объект (если вы используете Python 2.X, тогда взамен для поддержки такой возможности применяйте форму `**ключевые_аргументы`; за деталями обращайтесь в главу 18). После запуска этих функций выводятся следующие результаты вызова `zip` и двух вызовов дополняющих `map`:

```
[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]
```

Такие функции не поддаются переводу в списковые включения из-за того, что их циклы слишком специфичны. Тем не менее, хотя аналоги `zip` и `map` в текущий момент строят и возвращают результирующие списки, столь же легко посредством `yield` превратить их в *генераторы*, так что каждая функция станет возвращать по одной порции своего результирующего набора за раз. Результаты будут такими же, как раньше, но

нам понадобится снова использовать вызов `list`, чтобы заставить генераторы выдать свои значения для отображения:

```
# Применение генераторов: yield
def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    while all(seqs):
        yield tuple(S.pop(0) for S in seqs)
def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    while any(seqs):
        yield tuple((S.pop(0) if S else pad) for S in seqs)
S1, S2 = 'abc', 'xyz123'
print(list(myzip(S1, S2)))
print(list(mymapPad(S1, S2)))
print(list(mymapPad(S1, S2, pad=99)))
```

Наконец, ниже показана альтернативная реализация наших эмуляторов `zip` и `map` – вместо удаления аргументов из списков с помощью метода `pop` следующие версии делают свою работу путем вычисления минимальных и максимальных длин аргументов. Располагая длинами, легко написать вложенные списковые включения для прохода по диапазонам индексов аргументов:

```
# Альтернативная реализация с использованием длин
def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return [tuple(S[i] for S in seqs) for i in range(minlen)]
def mymapPad(*seqs, pad=None):
    maxlen = max(len(S) for S in seqs)
    index = range(maxlen)
    return [tuple((S[i] if len(S) > i else pad) for S in seqs) for i in index]
S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))
```

Поскольку в функциях применяется `len` и индексация, они предполагают, что аргументы являются последовательностями или чем-то подобным, а не произвольными итерируемыми объектами, как было в наших ранних версиях функций перемешивания и перестановки. Внешние включения выполняют проход по диапазонам индексов аргументов, а внутренние включения (передаваемые `tuple`) проходят по переданным последовательностям для параллельного извлечения аргументов. Результаты запуска будут такими же, как ранее.

Больше всего поражает в данном примере обилие генераторов и итераторов. Передаваемые в `min` и `max` аргументы представляют собой генераторные выражения, которые выполняются до конца перед тем, как начнется итерация по вложенным включениям. Более того, вложенные списковые включения задействуют два уровня отложенной оценки –строенная функция `range` из Python 3.X выдает итерируемый объект, как и аргумент в виде генераторного выражения для `tuple`.

На самом деле никакие результаты здесь не выпускаются до тех пор, пока квадратные скобки списковых включений не запросят значения для помещения в результатирующий список – они заставляют включения и генераторы выполняться. Чтобы пре-

вратить сами функции в генераторы вместо построителей списков, снова используйте круглые скобки вместо квадратных. Вот случай для `myzip`:

```
# Применение генераторов: (...)

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return (tuple(S[i] for S in seqs) for i in range(minlen))

S1, S2 = 'abc', 'xyz123'
print(list(myzip(S1, S2))) # Go!... [('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Для активации генераторов и других итерируемых объектов с целью выпуска ими результатов в этом случае требуется вызов `list`. Чтобы получить больше сведений, самостоятельно поэкспериментируйте с кодом. Разработка дальнейших альтернативных версий оставляется в качестве упражнения (см. также врезку “Что потребует внимания: одноразовые итерации” ниже).



Дополнительные примеры применения оператора `yield` предлагаются в главе 30, где он будет использоваться в сочетании с методом перегрузки операции `__iter__` для реализации определяемых пользователем итерируемых объектов в автоматическом режиме. Сохранение состояния локальных переменных в такой роли служит альтернативой атрибутам класса в том же духе, как функции замыканий из главы 17; однако, как будет показано, эта методика *комбинирует* классы и функциональные инструменты, а не выдвигает альтернативную парадигму.

Что потребует внимания: одноразовые итерации

В главе 14 мы видели, что некоторые встроенные функции (наподобие `map`) поддерживают одиночный обход, после чего опустошаются, и было обещано продемонстрировать пример того, как такой факт становится едва различимым, но важным на практике. Теперь, когда были проведены более глубокие исследования темы итерации, наступило время выполнить обещанное. Взгляните на следующую искусственную альтернативную версию кода примеров эмуляции `zip`, рассмотренных в настоящей главе, которая взята из руководств по Python:

```
def myzip(*args):
    iters = map(iter, args)
    while iters:
        res = [next(i) for i in iters]
        yield tuple(res)
```

Поскольку в коде применяются `iter` и `next`, он работает с итерируемым объектом любого типа. Обратите внимание, что нет никаких причин перехватывать исключение `StopIteration`, инициируемое `next(i)` внутри спискового включения, когда израсходуются элементы в любом итераторе из аргументов — это дает генераторной функции возможность дойти до конца и обеспечивает такой же эффект, как имел бы оператор `return`. Оператора `while iters:` достаточно для выполнения цикла, если был передан, по меньшей мере, один аргумент, и избегания бесконечного цикла в противном случае (списковое включение всегда возвращало бы пустой список).

Приведенный код нормально работает в Python 2.X в том виде, как есть:

```
>>> list(myzip('abc', 'lmnop'))
[('a', 'l'), ('b', 'm'), ('c', 'n')]
```

Но в Python 3.X он попадает в бесконечный цикл и терпит неудачу, потому что функция map из Python 3.X возвращает одноразовый итерируемый объект, а не список, как map из Python 2.X. В Python 3.X как только мы однократно выполним списковое включение внутри цикла, `iters` опустошается, но постоянно будет True (`a res = []`). Чтобы код заработал в Python 3.X, необходимо использовать встроенную функцию `list` для создания объекта, который способен поддерживать множество итераций:

```
def myzip(*args):
    iters = list(map(iter, args))      # Допускает множество просмотров
    ...остальной код не меняется...
```

Запустите код самостоятельно и отследите его работу. Урок здесь в том, что помещение вызовов map внутрь list в Python 3.X предназначено не только для отображения!

Сводка по синтаксису включений

Внимание в главе было сосредоточено на списковых включениях и генераторах, но имейте в виду, что в Python 3.X и 2.7 доступны еще две формы выражений с включениями: включения множеств и словарей. Мы кратко упоминали о них в главах 5 и 8, но благодаря вновь обретенным знаниям включений и генераторов вы уже должны быть в состоянии понять суть таких расширений.

- Для множеств новая литеральная форма `{1, 3, 2}` эквивалентна `set([1, 3, 2])`, а новый синтаксис включений множеств `{f(x) for x in S if P(x)}` подобен генераторному выражению `set(f(x) for x in S if P(x))`, где `f(x)` – произвольное выражение.
- Для словарей новый синтаксис включений словарей `{key: val for (key, val) in zip(keys, vals)}` работает подобно форме `dict(zip(keys, vals))`, а `{x: f(x) for x in items}` похож на генераторное выражение `dict((x, f(x)) for x in items)`.

Ниже представлена сводка по всем альтернативным версиям включений в Python 3.X и 2.7. Последние две версии недоступны в Python 2.6 и более ранних выпусках:

```
>>> [x * x for x in range(10)]          # Списковое включение: строит список
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]   # Подобно list(генераторное выражение)

>>> (x * x for x in range(10))        # Генераторное выражение:
                                           # производит элементы
<generator object at 0x009E7328>       # Круглые скобки часто необязательны

>>> {x * x for x in range(10)}        # Включение множества, Python 3.X и 2.7
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}   # {x, y} - тоже множество в этих версиях

>>> {x: x * x for x in range(10)}     # Включение словаря, Python 3.X и 2.7
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Области видимости и переменные включений

Теперь, когда были показаны все формы включений, вспомним приведенный в главе 17 обзор локализации переменных циклов в этих выражениях. Python 3.X локализует переменные циклов во всех четырех формах – временные имена переменных циклов в генераторах и включениях множеств, словарей и списков являются локальными в отношении выражения. Они не конфликтуют с именами вне выражения, но там не доступны и работают не так, как оператор цикла `for`:

```

c:\code> py -3
>>> (X for X in range(5))
<generator object <genexpr> at 0x00000000028E4798>
>>> X
NameError: name 'X' is not defined
Ошибка имени: имя X не определено

>>> X = 99
>>> [X for X in range(5)]      # Python 3.X: генераторы и включения множеств,
                                # словарей и списков локализуют
[0, 1, 2, 3, 4]
>>> X
99

>>> Y = 99
>>> for Y in range(5): pass    # Но операторы циклов имена не локализуют
>>> Y
4

```

Как упоминалось в главе 17, переменные Python 3.X, присвоенные во включении, на самом деле имеют особую вложенную область видимости; остальные имена, на которые производится ссылка внутри таких выражений, следуют обычным правилам LEGB. Например, в показанном ниже генераторе Z локализуется во включении, но Y и X находятся в объемлющей локальной и в глобальной областях видимости, как обычно:

```

>>> X = 'aaa'
>>> def func():
    Y = 'bbb'
    print('.join(Z for Z in X + Y))   # Z во включении, Y в локальной,
                                    # X в глобальной области видимости

>>> func()
aaabbb

```

В данном отношении ситуация *Python 2.X* аналогична, не считая того, что переменные *списковых включений* не локализуются — они работают в точности как в циклах *for* и сохраняют свои значения из последней итерации, но также подвержены неожиданным конфликтам с внешними именами. Включения множеств, словарей и генераторы локализуют имена как в *Python 3.X*:

```

c:\code> py -2
>>> (X for X in range(5))
<generator object <genexpr> at 0x0000000002147EE8>
>>> X
NameError: name 'X' is not defined
Ошибка имени: имя X не определено

>>> X = 99
>>> [X for X in range(5)]      # Python 2.X: подобно for списокное включение
                                # не локализует свои имена
[0, 1, 2, 3, 4]
>>> X
4

>>> Y = 99
>>> for Y in range(5): pass    # Циклы for не локализуют имена в Python 2.X и 3.X
>>> Y
4

```

Если вас заботит переносимость версий и гармония с оператором цикла `for`, тогда в качестве эмпирического правила применяйте уникальные имена для переменных в выражениях включений. Поведение Python 2.X имеет смысл с учетом того, что объект генератора отбрасывается после завершения производства результатов, но списковое включение эквивалентно циклу `for` — хотя эта аналогия не сохраняется для включений множеств и словарей, которые локализуют свои имена в обеих линейках Python и рассматриваются в следующем разделе.

Осмысление включений множеств и словарей

В известном смысле включения множеств и словарей представляют собой всего лишь синтаксический сахар для передачи генераторных выражений именам типов. Так как оба включения принимают любой итерируемый объект, показанные далее итераторы будут нормально работать:

```
>>> {x * x for x in range(10)}           # Включение
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> set(x * x for x in range(10))        # Генератор и имя типа
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> {x: x * x for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> dict((x, x * x) for x in range(10))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

>>> x          # Переменная цикла локализуется в Python 2.X и 3.X
NameError: name 'x' is not defined
Ошибка имени: имя x не определено
```

Тем не менее, что касается списковых включений, то мы всегда можем построить результирующие объекты также с помощью ручного кода. Вот эквиваленты, основанные на операторах, для последних двух включений (но согласно предыдущему разделу они отличаются локализацией имени):

```
>>> res = set()
>>> for x in range(10):      # Эквивалентное включение множества
    res.add(x * x)
>>> res
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> res = {}
>>> for x in range(10):      # Эквивалентное включение словаря
    res[x] = x * x
>>> res
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

>>> x          # Локализуется в выражениях включений, но не в операторах циклов
9
```

Обратите внимание, что хотя включения множеств и словарей принимают и просматривают итерируемые объекты, в них отсутствует понятие *генерации* результатов по запросу — обе формы строят сразу полные объекты. Если вы намереваетесь выпускать ключи и значения по запросу, тогда больше подойдет генераторное выражение:

```
>>> G = ((x, x * x) for x in range(10))
>>> next(G)
(0, 0)
>>> next(G)
(1, 1)
```

Расширенный синтаксис включений для множеств и словарей

Подобно списковым включениям и генераторным выражениям включения множеств и словарей поддерживают вложенные ассоциированные конструкции `if` для фильтрации элементов результата — в следующем примере накапливаются квадраты четных чисел (т.е. чисел, которые делятся на 2 без остатка) в диапазоне:

```
>>> [x * x for x in range(10) if x % 2 == 0]      # Списки упорядочиваются
[0, 4, 16, 36, 64]
>>> {x * x for x in range(10) if x % 2 == 0}      # Но множества - нет
{0, 16, 4, 64, 36}
>>> {x: x * x for x in range(10) if x % 2 == 0}    # Ключи словаря тоже
#     не упорядочиваются
{0: 0, 8: 64, 2: 4, 4: 16, 6: 36}
```

Вложенные циклы `for` также работают, хотя неупорядоченная и не допускающая дубликатов природа объектов обоих типов может сделать результаты менее прямолинейными для понимания:

```
>>> [x + y for x in [1, 2, 3] for y in [4, 5, 6]]  # Списки сохраняют дубликаты
[5, 6, 7, 6, 7, 8, 7, 8, 9]
>>> {x + y for x in [1, 2, 3] for y in [4, 5, 6]}  # Но множества - нет
{8, 9, 5, 6, 7}
>>> {x: y for x in [1, 2, 3] for y in [4, 5, 6]}  # Ключи словаря тоже
#     не сохраняют дубликаты
{1: 6, 2: 6, 3: 6}
```

Подобно списковым включениям разновидности включений множеств и словарей также могут выполнять проход по итерируемым объектам любого типа — спискам, строкам, файлам, диапазонам и всему остальному, что поддерживает протокол итерации:

```
>>> {x + y for x in 'ab' for y in 'cd'}
{'ac', 'bd', 'bc', 'ad'}
>>> {x + y: (ord(x), ord(y)) for x in 'ab' for y in 'cd'}
{'ac': (97, 99), 'bd': (98, 100), 'bc': (98, 99), 'ad': (97, 100)}
>>> {k * 2 for k in ['spam', 'ham', 'sausage'] if k[0] == 's'}
{'sausagesausage', 'spamspam'}
>>> {k.upper(): k * 2 for k in ['spam', 'ham', 'sausage'] if k[0] == 's'}
{'SAUSAGE': 'sausagesausage', 'SPAM': 'spamspam'}
```

Для лучшего освоения описанных инструментов поэкспериментируйте с ними самостоятельно. Они могут иметь либо не иметь преимущества в плане производительности перед альтернативами в виде генераторов или циклов `for`, но чтобы удостовериться в этом, нам нужно явно измерить время их выполнения — и мы естественным образом переходим к тематике следующей главы.

Резюме

Глава завершает наш обзор встроенных инструментов включений и итерации. Были исследованы списковые включения в контексте инструментов функционального программирования, а также представлены генераторные функции и выражения как дополнительные инструменты протокола итерации. Вдобавок мы подытожили четы-

ре формы включений в современном Python – генераторы плюс включения списков, множеств и словарей. Хотя были показаны все встроенные итерационные инструменты, мы еще вернемся к данной теме в главе 30 при изучении итерируемых объектов, которые основаны на определяемых пользователем классах.

Следующая глава является в некоторой степени продолжением темы пройденной главы – она заканчивает текущую часть книги реалистичным учебным примером, в котором измеряется производительность обсуждаемых здесь инструментов. Но прежде чем переходить к оценочным испытаниям включений и генераторов, закрепите пройденный материал этой главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. В чем разница между помещением спискового включения в квадратные скобки и в круглые скобки?
2. Как связаны между собой генераторы и итераторы?
3. Как определить, является ли функция генераторной?
4. Что делает оператор `yield`?
5. Как связаны между собой вызовы `map` и списковые включения? Сравните и противопоставьте их.

Проверьте свои знания: ответы

1. Списковые включения в квадратных скобках производят полный список в памяти за один раз. Когда списковые включения взамен помещаются в круглые скобки, то фактически являются генераторными выражениями – они имеют похожий смысл, но не выпускают сразу весь результирующий список. Генераторные выражения вместо этого возвращают объект генератора, который выдает по одному элементу результата за раз, когда используются в итерационном контексте.
2. Генераторы представляют собой итерируемые объекты, автоматически поддерживающие протокол итерации – они имеют итератор с методом `__next__` (`next` в Python 2.X), который многократно переходит на очередной элемент в серии результатов и инициирует исключение по достижении конца серии. В Python мы можем записывать генераторные функции с помощью `def` и `yield`, генераторные выражения посредством включений в круглых скобках и генераторные объекты с применением классов, в которых определен специальный метод по имени `__iter__` (обсуждается позже в книге).
3. Генераторная функция содержит в своем коде оператор `yield`. Во всем остальном генераторные функции синтаксически идентичны нормальным функциям, но они компилируются Python особым образом, чтобы при вызове возвращать итерируемый объект генератора. Такой объект сохраняет состояние и местоположение в коде между выдачей значений.
4. Когда оператор `yield` присутствует, он заставляет Python компилировать функцию специально как генераторную; при вызове функция возвращает объект генератора, который поддерживает протокол итерации. Когда оператор `yield` выполняется, он отправляет результат обратно вызывающему коду и приостанавливается с сохранением состояния функции; затем функция может возобновить

работу после последнего выполненного оператора `yield` в ответ на вызов встроенной функции `next` или метода `__next__`, инициированного вызывающим кодом. В более сложных ситуациях метод `send` генераторного объекта похожим образом возобновляет работу генератора, но способен также передавать значение, которое является значением выражения `yield`. Генераторные функции могут также содержать оператор `return`, прекращающий работу генератора.

5. Вызов `map` подобен списковому включению – оба они производят серию значений, накапливая результаты применения указанной операции к каждому элементу в последовательности или в другом итерируемом объекте по одному за раз. Главное отличие в том, что `map` применяет к каждому элементу вызов функции, а списковые включения применяют произвольные выражения. Из-за этого списковые включения более универсальны; они способны применять выражение вызова функции вроде `map`, а `map` требует функцию для применения выражений других видов. Списковые включения также поддерживают расширенный синтаксис, такой как вложенные циклы `for` и конструкции `if`, которые относятся к категории встроенной функции `filter`. В Python 3.X встроенная функция `map` также отличается тем, что производит *генератор* значений; списковое включение материализует сразу весь результирующий список в памяти. В Python 2.X оба инструмента создают результирующие списки.

Оценочные испытания

Теперь, когда известно, как писать код функций и пользоваться итерационными инструментами, мы собираемся приложить дополнительные усилия, чтобы заставить их работать вместе. Настоящая глава завершает часть книги, посвященную функциям, довольно крупным учебным примером, в котором измеряется относительная производительность описанных ранее итерационных инструментов.

Попутно в учебном примере исследуются инструменты Python для измерения времени выполнения кода, обсуждаются методики оценочных испытаний в целом и предоставляется возможность изучить код, который чуть более реалистичен и полезен, чем большинство того, что было показано до сих пор. Мы также измерим скорость текущих реализаций Python – величину, которая может быть, а может и не быть важной в зависимости от типа кода, который приходится писать.

Наконец, поскольку это последняя глава в данной части книги, мы представим обычный набор распространенных ловушек и упражнений, которые помогут приступить к реализации изученных идей. Но сначала давайте насладимся реалистичным приложением Python.

Измерение времени выполнения итерационных альтернатив

В книге встречалось несколько итерационных альтернатив. Как и многое другое в программировании, они демонстрируют компромиссы – в свете субъективных факторов вроде выразительности и более объективных критериев, таких как производительность. Часть вашей работы как программиста и инженера связана с выбором инструментов на основе факторов подобного рода.

С точки зрения производительности несколько раз упоминалось о том, что списковые включения иногда обладают преимуществом в скорости по сравнению с операторами цикла `for`, а вызовы `map` могут быть быстрее или медленнее в зависимости от схем вызова. Генераторные функции и выражения из предыдущей главы имеют тенденцию быть немного медленнее списковых включений, хотя они минимизируют требования относительно пространства памяти и не заставляют вызывающий код ожидать генерации всех результатов, когда их много.

В целом все это верно, но относительная производительность со временем может варьироваться из-за того, что внутренние механизмы Python постоянно изменяются и оптимизируются, а структура кода может произвольно влиять на скорость. Если вы хотите самостоятельно оценивать их производительность, тогда должны измерять время выполнения альтернативных версий на своем компьютере и своей версии Python.

Модуль измерения времени: любительский

К счастью, Python делает измерение времени выполнения кода простым. Например, чтобы получить суммарное время, необходимое для выполнения множества вызовов функции с произвольными позиционными аргументами, может быть достаточно следующей упрощенной функции:

```
# Файл timer0.py
import time
def timer(func, *args):          # Упрощенная функция измерения времени
    start = time.clock()
    for i in range(1000):
        func(*args)
    return time.clock() - start  # Суммарное истекшее время в секундах
```

Функция `timer` работает – она извлекает значения времени из модуля Python по имени `time` и вычитает системное время начала из времени останова после выполнения 1 000 вызовов переданной функции с переданными инструментами. Вот пример ее запуска на компьютере с версией Python 3.7:

```
>>> from timer0 import timer
>>> timer(pow, 2, 1000)         # Время, затраченное на вызов pow(2, 1000)
1000 раз
0.0029763490000007664
>>> timer(str.upper, 'spam')   # Время, затраченное на вызов 'spam'.upper()
1000 раз
0.00017862699999682263
```

Наряду с простотой функция `timer` также является довольно ограниченной и намеренно демонстрирует несколько классических заблуждений относительно проектирования функций и оценочных испытаний. Помимо прочего она:

- не поддерживает ключевые аргументы в вызове тестируемой функции;
- имеет жестко закодированное количество повторений;
- добавляет затраты на вызов `range` ко времени тестируемой функции;
- всегда использует метод `time.clock`, который может оказаться не самым лучшим вариантом за пределами Windows;
- не предоставляет вызывающему коду способа проверки, что тестируемая функция действительно работает;
- дает только суммарное время, которое может меняться на сильно загруженных компьютерах.

Другими словами, код измерения времени сложнее, чем вы могли ожидать! Чтобы обеспечить большую универсальность и точность, давайте расширим его до по-прежнему простых, но более полезных служебных функций хронометраж, которые мы сможем задействовать для сравнения итерационных альтернатив и применять для решения других задач измерения времени в будущем. Функции будут помещены в файл модуля, так что их можно будет использовать в разнообразных программах, и иметь строки документации, предоставляющие основные детали, которые инструмент PyDoc способен отображать по запросу – на рис. 15.2 был представлен экранный снимок страниц документации, визуализируемых для создаваемых здесь модулей измерения времени:

```

# Файл timer.py
"""
Homegrown timing tools for function calls.
Does total time, best-of time, and best-of-totals time
Любительские инструменты для измерения времени выполнения вызовов функций.
Определяют суммарное время, лучшее время и лучшее суммарное время
"""

import time, sys
timer = time.clock if sys.platform[:3] == 'win' else time.time
def total(reps, func, *pargs, **kargs):
    """
    Total time to run func() reps times.
    Returns (total time, last result)
    Суммарное время выполнения функции func() reps раз
    Возвращает (суммарное время, последний результат)
    """
    repslist = list(range(reps)) # Вынести за пределы, уравнять Python 2.x, 3.x
    start = timer() # Или perf_counter/другая в Python 3.3+
    for i in repslist:
        ret = func(*pargs, **kargs)
        elapsed = timer() - start
    return (elapsed, ret)

def bestof(reps, func, *pargs, **kargs):
    """
    Quickest func() among reps runs.
    Returns (best time, last result)
    Самая быстрая функция func() среди reps запусков.
    Возвращает (лучшее время, последний результат)
    """
    best = 2 ** 32 # 136 лет представляется достаточным
    for i in range(reps): # Использование range при измерении времени не учитывается
        start = timer()
        ret = func(*pargs, **kargs)
        elapsed = timer() - start # Или вызвать total() с reps=1
        if elapsed < best: best = elapsed # Или добавить в список и найти min()
    return (best, ret)

def bestoftotal(reps1, reps2, func, *pargs, **kargs):
    """
    Best of totals:
    (best of reps1 runs of (total of reps2 runs of func))
    Лучшее суммарное время:
    (лучшее время из reps1 запусков (суммарное время reps2 запусков func))
    """
    return bestof(reps1, total, reps2, func, *pargs, **kargs)

```

В модуле timer реализованы функции измерения *суммарного* (total) и *лучшего* (bestof) времени выполнения вызовов, а также функция измерения *лучшего суммарного* (bestoftotal) времени, которая комбинирует первые две функции. В каждой измеряется время выполнения вызова любой функции с любыми позиционными и ключевыми аргументами, переданными по отдельности, за счет извлечения времени начала, вызова функции и вычитания времени начала из времени окончания.

Ниже объясняется, каким образом данная реализация устраниет недостатки предшествующей версии.

- Модуль `time` в Python предоставляет доступ к текущему времени с точностью, которая зависит от платформы. В Windows заявлено, что его функция `clock` обеспечивает микросекундную степень детализации и потому очень точна. Поскольку функция `time` в Unix может быть лучше, сценарий автоматически делает выбор между ними на основе строки `platform` из модуля `sys`; при выполнении в среде Windows она начинается с `'win'`. Другие варианты в Python 3.3 и последующих версиях, которые по причине переносимости здесь не применяются, описаны во врезке “Новые вызовы таймеров, появившиеся в версии Python 3.3” далее в главе. Дело в том, что мы будем также измерять время в Python 2.X, где более новые вызовы не доступны, и их результаты на платформе Windows в любом случае выглядят похожими в Python 3.3 и последующих версиях.
- Вызов `range` в функции `total` вынесен за пределы цикла измерения времени, поэтому связанные с ним затраты не учитываются для хронометрируемой функции в Python 2.X. Вызов `range` в Python 3.X дает итерируемый объект, так что данный шаг необязателен и безвреден, но мы все равно прогоняем результат через `list` и затраты на обход оказываются одинаковыми в Python 2.X и 3.X. Сказанное неприменимо к функции `bestof`, т.к. время, связанное с `range`, не учитывается при измерении времени выполнения тестируемой функции.
- Счетчик `reps` передается в качестве аргумента перед тестируемой функцией и ее аргументами, чтобы позволить варьировать количество повторений для каждого вызова.
- Любое количество позиционных и ключевых аргументов собирается посредством синтаксиса аргументов со звездочкой, так что они должны отправляться индивидуально, а не внутри последовательности или словаря. При необходимости вызывающий код может распаковывать коллекции аргументов в отдельные аргументы с помощью звездочек в вызове, как делается в конце функции `bestoftotal`. Такой прием обсуждался в главе 18.
- Первая функция в модуле возвращает суммарное истекшее время для всех вызовов в кортеже вместе с финальным возвращаемым значением измеряемой функции, чтобы в вызывающем коде была возможность проверить правильность ее выполнения.
- Вторая функция делает похожую работу, но вместо суммарного возвращает лучшее (минимальное) время – более полезное, когда желательно отфильтровать влияние другой активности на компьютере, и менее полезное для тестируемых функций, которые выполняются слишком быстро, чтобы давать значимые показатели времени прогона.
- Для решения проблемы, упомянутой в предыдущем пункте, последняя функция модуля запускает вложенные вызовы `total` внутри `bestof`, чтобы получить лучшее суммарное время. Вложенная операция `total` способна сделать показатели времени прогона более полезными, но мы по-прежнему имеем фильтр для лучшего времени. Код функции может быть легче понять, если помнить о том, что каждая функция является передаваемым объектом, даже сами тестирующие функции.

В более широком плане, поскольку описанные функции реализованы в файле модуля, они становятся универсально полезными инструментами везде, где возникнет желание их импортировать. Модули и импортование были представлены в главе 3 и более подробно рассматриваются в следующей части книги, а пока просто импортируйте модуль и вызывайте необходимые функции измерения времени. В несложных ситуациях модуль дает схожий с предшественником результат, но будет гораздо надежнее в более крупных контекстах. Вот пример вызовов функций модуля `timer` в Python 3.7:

```
>>> import timer
>>> timer.total(1000, pow, 2, 1000)[0]      # Сравнимо с показанными ранее
                                                # результатами timer0
0.0029486640000015996
>>> timer.total(1000, str.upper, 'spam')     # Возвращает (время, результат
                                                # последнего вызова)
(0.0001786260000002926, 'SPAM')
>>> timer.bestof(1000, str.upper, 'spam')     # Почти 1/1555 часть
                                                # суммарного времени
(1.8960000005563415e-06, 'SPAM')
>>> timer.bestof(1000, pow, 2, 1000000)[0]
0.017638512999994305
>>> timer.bestof(50, timer.total, 1000, str.upper, 'spam')
(0.00020517400000130692, (0.00017483400000628535, 'SPAM'))
>>> timer.bestoftotal(50, 1000, str.upper, 'spam')
(0.0002047939999982873, (0.00017445399998905486, 'SPAM'))
```

Последние два вызова измеряют *лучшее суммарное время* – наименьшее время из 50 прогонов, каждый из которых измеряет суммарное время выполнения 1000 вызовов `str.upper` (приблизительно соответствующее показаниям времени в начале листинга). Функция, используемая в последнем вызове, на самом деле является всего лишь удобством, которое отображается на предшествующую ей форму вызова; обе они возвращают кортеж с лучшим временем, умещающий в себе кортеж с суммарным временем и результатом последнего вызова.

Сравните последние два результата со следующей альтернативой, основанной на генераторе:

```
>>> min(timer.total(1000, str.upper, 'spam') for i in range(50))
(0.00017445399998905486, 'SPAM')
```

Применение `min` к итерации по результатам `total` таким способом обеспечивает похожий эффект, потому что сравнениями, выполняемыми `min`, управляют показания времени в результирующих кортежах (они крайние слева в кортежах). Мы могли бы использовать этот прием в своем модуле (и поступим так в более поздних вариациях); он слегка варьируется, пренебрегая очень малыми издержками в коде функции `bestof` и не вкладывая результирующие кортежи, но для относительных сравнений вполне достаточно любого из двух результатов. В том виде, как есть, функция `bestof` должна выбрать высокое начальное значение наименьшего показания времени – хотя 136 лет представляется гораздо более длительным промежутком, чем будет выполнятьсь большинство тестов!

```
>>> (((2 ** 32) / 60) / 60) / 24) / 365      # Плюс еще несколько добавочных дней
136.19251953323186
>>> (((2 ** 32) // 60) // 60) // 24) // 365      # Округление в меньшую
                                                # сторону: см. главу 5
```

Новые вызовы таймеров, появившиеся в версии Python 3.3

В этом разделе задействованы вызовы `clock` и `time` из модуля `time`, т.к. они подходят всем читателям книги. В Python 3.3 в модуле `time` появились новые интерфейсы, которые спроектированы как более переносимые. В частности, поведение вызовов `clock` и `time` данного модуля меняется в зависимости от платформы, но его новые функции `perf_counter` и `process_time` имеют четко определенную и нейтральную к платформе семантику.

- Функция `time.perf_counter()` возвращает значение в дробных секундах счетчика производительности, определенного как часы с наивысшим доступным разрешением для измерения коротких промежутков времени. Она включает время, истекшее в состояниях сна, и действует в масштабе всей системы.
- Функция `time.process_time()` возвращает значение в дробных секундах суммы системного и пользовательского процессорного времени текущего процесса. Она не включает время, истекшее в состояниях сна, и по определению действует в масштабе процесса.

Для обоих вызовов начало отсчета возвращаемого значения не определено, так что имеет силу только *разница* между результатами последовательных вызовов. Вызов `perf_counter` можно воспринимать как фактическое время, а начиная с версии Python 3.8, он по умолчанию применяется для оценочных испытаний в обсуждаемом далее модуле `timeit`; вызов `process_time` дает процессорное время переносимым образом.

Как показано в книге, в наши дни вызов `time.clock` все еще годен к употреблению в среде Windows. В руководствах по Python 3.3 он документирован как устаревший, но никаких предупреждений в случае его использования не выдается — теперь его планируют удалить в версии Python 3.8. При необходимости вы можете опознавать Python 3.3 или последующую версию с помощью приведенного ниже кода, который я решил не применять ради краткости и сопоставимости таймеров:

```
if sys.version_info[0] >= 3 and sys.version_info[1] >= 3:  
    timer = time.perf_counter      # или process_time  
else:  
    timer = time.clock if sys.platform[:3] == 'win' else time.time
```

В качестве альтернативы следующий код также добавит переносимость и защитит от будущего устаревания, хотя он затрагивает тему исключений, которая пока еще не раскрывалась в полной мере, и его выборы способны сделать недействительными сравнения скоростей выполнения между версиями — таймеры могут отличаться по своему разрешению!

```
try:  
    timer = time.perf_counter      # или process_time  
except AttributeError:  
    timer = time.clock if sys.platform[:3] == 'win' else time.time
```

Если бы я писал книгу в расчете на читателей, работающих лишь с Python 3.3+, то использовал бы новые и очевидно улучшенные вызовы, и вы сами должны поступать так, когда находитесь в подобном положении. Однако более новые вызовы не будут работать у пользователей любых других выпусков Python, которых по-прежнему большинство в современном мире Python. Да, легче сделать вид, что прошлое не имеет значения, но это было бы не только бегством от реальности, но и просто явным неуважением.

Сценарий измерения времени

Чтобы измерить скорость итерационных инструментов (наша исходная цель), мы запустим следующий сценарий – в нем применяется модуль `timer`, который был написан для измерения относительных скоростей изученных ранее методик построения списков:

```
# Файл timeseqs.py
# Проверка относительной скорости итерационных альтернатив.

import sys, timer                                # Импортирование функций timer
reps = 10000                                       # Вынесение наружу, список
repplist = list(range(reps))                      #   в Python 2.X/3.X

def forLoop():
    res = []
    for x in repplist:
        res.append(abs(x))
    return res

def listComp():
    return [abs(x) for x in repplist]

def mapCall():
    return list(map(abs, repplist))                # Использовать list() только
                                                    #   в Python 3.X!
    # return map(abs, repplist)

def genExpr():
    return list(abs(x) for x in repplist)          # list() требуется для инициирования
                                                    #   выпуска результатов

def genFunc():
    def gen():
        for x in repplist:
            yield abs(x)
    return list(gen())                            # list() требуется для инициирования выпуска результатов

print(sys.version)
for test in (forLoop, listComp, mapCall, genExpr, genFunc):
    (bestof, (total, result)) = timer.bestoftotal(5, 1000, test)
    print ('%-9s: %.5f => [%s...%s]' %
          (test.__name__, bestof, result[0], result[-1]))
```

В сценарии проверяются пять альтернативных способов создания списков результатов. Как видно, сообщаемые им показания времени отражают порядка 10 миллионов шагов для каждой из пяти тестируемых функций – каждая строит список из 10 000 элементов 1000 раз. Процесс повторяется 5 раз, чтобы получить лучшее время для каждой из 5 тестируемых функций, суммарно давая колоссальное количество в 250 миллионов шагов сценария (впечатляющее, но вполне реально на большинстве компьютеров в наши дни).

Обратите внимание на то, что мы должны прогнать результаты генераторного выражения и функции через вызов встроенного метода `list`, заставляя их выдавать все свои значения. Если этого не сделать, тогда в Python 2.X и 3.X мы произвели бы просто генераторы, не делающие какую-либо реальную работу. В Python 3.X мы обязаны делать то же самое для результата `map`, т.к. теперь он также является итерируемым объектом. Для Python 2.X вызов `list` вокруг `map` потребуется удалить вручную, чтобы

избежать добавления излишних накладных расходов на создание списка к тестируемой функции (хотя в большинстве тестов его влияние пренебрежимо мало).

Аналогичным образом результат `range` внутреннего цикла выносится в начало модуля, чтобы убрать затраты на его создание из суммарного времени, и помещается внутрь вызова `list`, а потому его затраты на обход не будут отклоняться из-за того, что он представляет собой генератор в Python 3.X (почти так же, как мы поступали в модуле `timer`). Результат может быть омрачен затратами внутреннего цикла итераций, но лучше удалить как можно больше переменных.

Кроме того, взгляните, как код в конце проходит по кортежу из пяти объектов функций и выводит атрибут `_name_` каждого: вам уже известно, что это встроенный атрибут, который дает имя функции¹.

Результаты измерения времени

Запустив сценарий из предыдущего раздела в *Python 3.7* на компьютере Windows 7, мы получаем следующие результаты — `map` быстрее списковых включений, они оба быстрее циклов `for`, а генераторные выражения и функции находятся посередине (показания времени здесь представляют собой суммарное время в секундах):

```
C:\code> c:\python37\python timeseqs.py
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
forLoop : 2.34160 => [0...9999]
listComp : 1.31783 => [0...9999]
mapCall : 0.41968 => [0...9999]
genExpr : 1.92063 => [0...9999]
genFunc : 1.93672 => [0...9999]
```

Если вы достаточно долго исследуете код сценария `timeseqs.py` и его вывод, то заметите, что в наши дни генераторные выражения выполняются медленнее списковых включений. Несмотря на то что помещение генераторного выражения внутрь вызова `list` делает его функционально эквивалентным списковому включению в квадратных скобках, *внутренние* реализации двух выражений видимо отличаются (правда, мы фактически учитываем для генератора также и время вызова `list`):

```
return [abs(x) for x in repslist]      # 1.32 секунды
return list(abs(x) for x in repslist)    # 1.92 секунды: внутренне отличаются
```

Хотя выяснение точной причины потребовало бы более глубокого анализа (и возможно изучения исходного кода), похоже, это имеет смысл, учитывая тот факт, что генераторное выражение должно делать дополнительную работу по сохранению и восстановлению своего состояния во время выпуска значений. Списковое включение в такой работе не нуждается и выполняется несколько быстрее здесь и в более поздних тестах.

Интересно отметить, что когда я запускал сценарий `timeseqs.py` в версии Python 3.0 на компьютере Windows Vista для четвертого издания книги и в версии Python 2.5

¹ Предварительный обзор: обратите внимание, что здесь мы обязаны передавать функции в `timer` вручную. В главах 39 и 40 второго тома мы увидим альтернативные таймеры на основе *декораторов*, посредством которых хронометрируемые функции вызываются обычным образом, но требуют дополнительного синтаксиса @ в местах, где они определяются. Декораторы могут быть более удобными для снабжения функций логикой измерения времени, когда они уже используются внутри крупной системы, и они не настолько легко поддерживают предполагаемые здесь более изолированные схемы вызова тестов. Если функция декорирована, тогда *каждый* ее вызов запускает логику измерения времени, что в зависимости от преследуемых целей может быть плюсом или минусом.

на компьютере Windows XP для третьего издания, результаты были относительно похожими – списковые включения оказались в два раза быстрее эквивалентных операторов цикла `for`, а `map` чуть быстрее списковых включений при отображении встроенной функции `abs` (модуль числа). Абсолютные показания времени в Python 2.5 были примерно в четыре-пять раз больше, чем в текущем выводе Python 3.7, но вероятно на результаты больше повлиял высокопроизводительный компьютер, нежели какие-либо усовершенствования в Python.

На самом деле почти все результаты в Python 2.7 для данного сценария оказываются более быстрыми, чем в Python 3.7 – вызов `list` из `map` был удален во избежание двукратного создания списка результатов, но даже если его оставить, то он увеличит показание времени на небольшую постоянную величину:

```
c:\code> c:\python27\python timeseqs.py
2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)]
forLoop : 1.24902 => [0...9999]
listComp : 0.66970 => [0...9999]
mapCall : 0.57018 => [0...9999]
genExpr : 0.90339 => [0...9999]
genFunc : 0.90542 => [0...9999]
```

Для сравнения ниже приведены результаты измерения скорости выполнения тех же тестов в PyPy – оптимизированной реализации Python, обсуждавшейся в главе 2, версия 1.9 которой реализует язык Python 2.7. Здесь версия PyPy 1.9 почти в 10 раз (на порядок) быстрее; она покажет себя даже еще лучше, когда мы возвратимся к сравнению версий Python позже в главе, применяя инструменты с отличающимися кодовыми структурами (правда, PyPy также и отстает на ряде других тестов):

```
c:\code> c:\PyPy\pypy-1.9\pypy.exe timeseqs.py
2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit]
forLoop : 0.10106 => [0...9999]
listComp : 0.05629 => [0...9999]
mapCall : 0.10022 => [0...9999]
genExpr : 0.17234 => [0...9999]
genFunc : 0.17519 => [0...9999]
```

В самой реализации PyPy списковые включения быстрее `map`, но более важным представляется тот факт, что все результаты PyPy оказались настолько быстрее. В CPython функция `map` до сих пор была самой быстрой.

Влияние вызовов функций: `map`

Тем не менее, давайте посмотрим, что произойдет, если мы изменим сценарий с целью выполнения на каждой итерации внутристрочной операции, такой как сложение, а не вызова встроенной функции `abs` (опущенные части кода остались неизмененными, а вызов `list` вокруг `map` нужен только для Python 3.7):

```
# Файл timeseqs2.py (отличающиеся части)
...
def forLoop():
    res = []
    for x in repslist:
        res.append(x + 10)
    return res

def listComp():
    return [x + 10 for x in repslist]
```

```

def mapCall():
    return list(map((lambda x: x + 10), repslist))      # list() только в Python
3.X

def genExpr():
    return list(x + 10 for x in repslist)                # list() в Python 2.X +
3.X

def genFunc():
    def gen():
        for x in repslist:
            yield x + 10
    return list(gen())                                    # list() в Python 2.X + 3.X
...

```

Теперь необходимость вызывать определяемую пользователем функцию для вызова `map` делает его медленнее операторов цикла `for` вопреки тому факту, что версия с операторами цикла крупнее в плане кода. Другими словами, удаление вызовов функций может сделать остальные версии более быстрыми (подробнее об этом в предстоящей врезке “На заметку!”). Вот результаты в Python 3.7:

```
c:\code> c:\python37\python timeseqs2.py
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
forLoop : 1.78581 => [10...10009]
listComp : 0.90618 => [10...10009]
mapCall : 1.95757 => [10...10009]
genExpr : 1.63313 => [10...10009]
genFunc : 1.63656 => [10...10009]
```

Результаты в CPython также были согласованными. Результаты в Python 3.0 для предыдущего издания книги, полученные на менее производительном компьютере, снова оказались относительно похожими, хотя почти вдвое медленнее из-за отличий компьютеров, где проводилось тестирование (результаты в Python 2.5 снова были в четыре-пять раз медленнее по сравнению с текущими результатами).

Из-за того, что интерпретатор внутренне производит настолько много оптимизаций, анализ производительности кода Python такого вида является крайне сложным делом. Однако без цифр практически невозможно предугадать, какой метод будет выполняться наилучшим образом; все, что вы можете сделать – измерить время выполнения собственного кода на своем компьютере с имеющейся версией Python.

В нашем случае мы можем сказать наверняка лишь то, что в данной версии Python использование определяемой пользователем функции в вызовах `map`, похоже, существенно снижает производительность (хотя операция `+` также может быть медленнее тривиальной функции `abs`), и то, что списковые включения выполняются быстрее (хотя медленнее `map` в других ситуациях). Списковые включения согласованно выглядят в два раза более быстрыми, чем циклы `for`, но даже здесь требуется уточнение – на относительную скорость спискового включения могут повлиять применение дополнительного синтаксиса (например, фильтров `if`), изменения в Python и режимы использования, которые мы не хронометрировали.

Тем не менее, как упоминалось ранее, при написании кода Python производительность не должна быть вашей главной задачей; первое, что вы обязаны делать для оптимизации кода Python – не оптимизировать его! Пишите код изначально с акцентом на читабельности и простоте, а затем позже оптимизируйте, если и только если в этом есть необходимость. Вполне возможно, что любая из пяти альтернатив окажется достаточно быстрой для наборов данных, которые вашей программе нужно обрабатывать; в таком случае главной целью должна быть понятность программы.



Чтобы копнуть еще глубже, модифицируйте код для применения простой определяемой пользователем функции во всех пяти хронометрируемых методиках итерации. Скажем, поступите так (код взят из файла `timeseqs2B.py` в пакете примеров):

```
def F(x): return x
def listComp():
    return [F(x) for x in repslist]
def mapCall():
    return list(map(F, repslist))
```

Результаты (приведенные в файле `timeseqs-results.txt`) относительно похожи на использование встроенной функции `abs` — по крайней мере, в CPython функция `map` оказывается самой быстрой. В более общем смысле среди пяти методик итерации на сегодняшний день `map` будет самой быстрой, если во всех пяти вызывается любая *функция*, встроенная или нет, но самой медленной, когда остальные этого не делают.

То есть `map` кажется медленнее всего лишь *из-за того, что она требует вызовов функций*, а вызовы функций являются относительно медленными вообще. Поскольку `map` не способна избежать вызова функций, она может проиграть просто по ассоциации! Другие итерационные инструменты выигрывают оттого, что они в состоянии действовать без вызовов функций. Мы подтвердим это открытие в предстоящих тестах, выполняемых с применением модуля `timeit`.

Альтернативные версии модуля для измерения времени

Модуль для измерения времени из предыдущего раздела нормально работает, но он мог бы быть чуть более дружественным к пользователю. Совершенно очевидно, что его функции требуют передачи счетчика повторений как первого аргумента и не предлагают для него какого-то стандартного значения — возможно мелочь, но далеко не идеальное решение в универсальном инструменте. Мы также могли бы задействовать показанную ранее методику с `min`, чтобы немного упростить возвращаемое значение и убрать из результирующего времени небольшие накладные расходы.

Ниже приведена альтернативная реализация модуля `timer`, которая решает указанные задачи, позволяя передавать счетчик повторений в виде ключевого аргумента по имени `_reps`:

```
# Файл timer2.py (Python 2.X и 3.X)
"""
total(spam, 1, 2, a=3, b=4, _reps=1000) вызывает и хронометрирует spam(1, 2, a=3, b=4)
_reps раз и возвращает суммарное время для всех прогонов с финальным результатом.

bestof(spam, 1, 2, a=3, b=4, _reps=5) запускает тест лучшего из N в попытке
избавиться от влияния колебаний загрузки системы и возвращает лучшее время
среди _reps тестов.

bestofftotal(spam, 1, 2, a=3, b=4, _reps1=5, _reps=1000) запускает тест
лучшего суммарного времени, который берет лучший из _reps1 прогонов
(суммарного времени _reps прогонов);
"""

import time, sys
timer = time.clock if sys.platform[:3] == 'win' else time.time

def total(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 1000)      # Переданное или стандартное
                                            # количество повторений
```

```

repslist = list(range(_reps))    # Вынести range наружу для списков Python 2.X
start = timer()
for i in repslist:
    ret = func(*pargs, **kargs)
elapsed = timer() - start
return (elapsed, ret)

def bestof(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 5)
    best = 2 ** 32
    for i in range(_reps):
        start = timer()
        ret = func(*pargs, **kargs)
        elapsed = timer() - start
        if elapsed < best: best = elapsed
    return (best, ret)

def bestoftotal(func, *pargs, **kargs):
    _reps1 = kargs.pop('_reps1', 5)
    return min(total(func, *pargs, **kargs) for i in range(_reps1))

```

Строки документации в начале файла модуля `timer2.py` описывают планируемое использование. С применением словарных операций поп аргумент `_reps` удаляется из аргументов, предназначенных для тестируемой функции, и снабжается стандартным значением (аргумент имеет необычное имя, чтобы избежать конфликта с реальными ключевыми аргументами, относящимися к хронометрируемой функции).

Обратите внимание, что функция `bestoftotal` здесь вместо вложенных вызовов использует показанную ранее схему с `min` и генератором, отчасти из-за того, что это упрощает результаты и устраниет небольшой расход времени, который был в предыдущей версии (где код извлекал лучшее время *после* измерения суммарного времени), но также и потому, что она обязана поддерживать два разных ключевых аргумента для повторений со стандартными значениями — `total` и `bestof` не могут применять одно и то же имя аргумента. Добавьте в код вывод аргументов, если он поможет отследить его работу.

Чтобы задействовать новый модуль для измерения времени, можно следующим образом изменить сценарий или обратиться к готовому файлу `timeseqs_timer2.py` в коде примеров; результаты будут по существу теми же, что и ранее (изменения касаются только API-интерфейса), поэтому здесь они не показаны:

```

import sys, timer2
...
for test in (forLoop, listComp, mapCall, genExpr, genFunc):
    (total, result) = timer2.bestoftotal(test, _reps1=5, _reps=1000)

# Или:
#     (total, result) = timer2.bestoftotal(test)
#     (total, result) = timer2.bestof(test, _reps=5)
#     (total, result) = timer2.total(test, _reps=1000)
#     (bestof, (total, result)) = timer2.bestof(timer2.total, test, _reps=5)

print ('%-9s: %.5f => [%s...%s]' %
      (test.__name__, total, result[0], result[-1]))

```

Можно также запустить несколько интерактивных тестов, как делалось для первоначальной версии — результаты в Python 3.7 снова оказываются похожими, но мы передаем счетчики повторений в виде ключевых аргументов и получаем стандартные значения, если аргументы не указаны:

```

>>> from timer2 import total, bestof, bestoftotal
>>> total(pow, 2, 1000) [0]           # 2 ** 1000, по умолчанию 1K повторений
0.002912668999969813
>>> total(pow, 2, 1000, _reps=1000) [0]      # 2 ** 1000, 1K повторений
0.002941492000050415
>>> total(pow, 2, 1000, _reps=1000000) [0]    # 2 ** 1000, 1M повторений
3.0047937669999953
>>> bestof(pow, 2, 100000) [0]        # 2 ** 100K, по умолчанию 5 повторений
0.001231056999840482
>>> bestof(pow, 2, 1000000, _reps=30) [0]     # 2 ** 1M, лучшее из 30
0.01770887500001095
>>> bestoftotal(str.upper, 'spam', _reps1=30, _reps=1000)   # Лучшее из 30,
                                                               # суммарное из 1K
(0.0001740769999969416, 'SPAM')
>>> bestof(total, str.upper, 'spam', _reps=30) # Вложенные вызовы тоже работают
(0.00020669299999553914, (0.00017521499998451873, 'SPAM'))

```

Чтобы взглянуть, как теперь поддерживаются ключевые аргументы, определим функцию с большим количеством аргументов и передадим некоторые по имени:

```

>>> def spam(a, b, c, d): return a + b + c + d
>>> total(spam, 1, 2, c=3, d=4, _reps=1000)
(0.0004759640000031595, 10)
>>> bestof(spam, 1, 2, c=3, d=4, _reps=1000)
(2.276000003575973e-06, 10)
>>> bestoftotal(spam, 1, 2, c=3, d=4, _reps1=1000, _reps=1000)
(0.00046306799998774295, 10)
>>> bestoftotal(spam, *(1, 2), _reps1=1000, _reps=1000, **dict(c=3, d=4))
(0.0004585180000162836, 10)

```

Использование аргументов с передачей только по ключевым словам в Python 3.X

И последний момент: чтобы упростить код модуля для измерения времени, мы также можем применять *аргументы с передачей только по ключевым словам* Python 3.X. Как выяснилось в главе 18, аргументы с передачей только по ключевым словам идеально подходят для конфигурационных параметров, таких как аргумент `_reps` наших функций. Они должны записываться после `*` и перед `**` в заголовке функции, а в вызове функции обязаны передаваться по ключевому слову и находиться перед `**` при их наличии. Ниже представлена альтернативная версия предыдущего модуля, использующая аргументы с передачей только по ключевым словам. Несмотря на более простой вид, она компилируется и выполняется только в Python 3.X, но не Python 2.X:

```
# Файл timer3.py (только Python 3.X)
"""

```

Используется в точности как `timer2.py`, но для получения более простого кода применяет аргументы с передачей только по ключевым словам и стандартными значениями Python 3.X.

Выносить вызов `range()` за пределы тестов в Python 3.X нет нужды, т.к. он всегда дает генератор; данная версия не будет работать в Python 2.X.

```

import time, sys
timer = time.clock if sys.platform[:3] == 'win' else time.time

```

```

def total(func, *pargs, _reps=1000, **kargs):
    start = timer()
    for i in range(_reps):
        ret = func(*pargs, **kargs)
    elapsed = timer() - start
    return (elapsed, ret)

def bestof(func, *pargs, _reps=5, **kargs):
    best = 2 ** 32
    for i in range(_reps):
        start = timer()
        ret = func(*pargs, **kargs)
        elapsed = timer() - start
        if elapsed < best: best = elapsed
    return (best, ret)

def bestoftotal(func, *pargs, _reps1=5, **kargs):
    return min(total(func, *pargs, **kargs) for i in range(_reps1))

```

Версия `timer3.py` применяется таким же способом, как предыдущая, и производит идентичные результаты, поэтому здесь не имеет смысла повторять вывод тех же самых тестов; при желании поэкспериментируйте с модулем `timer3.py` самостоятельно. Уделите внимание правилам упорядочения аргументов в вызовах. Например, вот как вызывать функцию `bestof`, которая запускает `total`:

```
(elapsed, ret) = total(func, *pargs, _reps=1, **kargs)
```

Аргументы с передачей только по ключевым словам в Python 3.X обсуждались в главе 18; они способны упростить код конфигурируемых инструментов, подобных функциям из модуля `timer3.py`, но не имеют обратной совместимости с Python 2.X. Если вы хотите сравнивать скорости в Python 2.X и Python 3.X или обеспечить для программистов возможность использования любой из двух линеек Python, тогда предыдущая версия модуля (`timer2`) вероятно будет наилучшим выбором.

Также имейте в виду, что для тривиальных функций вроде некоторых из тестируемых с помощью предыдущей версии модуля, затраты кода измерения времени иногда могут быть настолько же существенными, как затраты простых хронометрируемых функций, поэтому вы не должны воспринимать результаты хронометражка чересчур буквально. Однако результаты хронометражка могут помочь в оценке относительных скоростей выполнения кодовых альтернатив и быть более значащими для операций, которые выполняются дольше или повторяются чаще.

Другие варианты

Чтобы получить больше информации, попробуйте модифицировать счетчики повторений, применяемые этими модулями, либо исследуйте альтернативный модуль `timeit` из стандартной библиотеки Python, который автоматизирует хронометраж кода, поддерживает режим использования в командной строке и обходит специфичные к платформам проблемы — на самом деле мы задействуем его в следующем разделе.

У вас также может возникнуть желание взглянуть на стандартный библиотечный модуль `profile`, представляющий собой законченный инструмент для профилирования исходного кода. Мы будем его исследовать в главе 36 в контексте инструментов разработки для крупных проектов. В целом вы должны профилировать код с целью изоляции узких мест еще до переписывания и хронометража кодовых альтернатив, как делось здесь.

Вы можете попробовать изменить или эмулировать сценарий хронометража для измерения скорости *включений множеств и словарей* Python 3.X и 2.7, показанных в предыдущей главе, вместе с их эквивалентами в форме циклов `for`. В программах на Python они применяются реже, чем построение списков результатов, так что мы оставляем эту задачу в качестве упражнения для самостоятельного выполнения (пожалуйста, не спорьте...); следующий раздел частично испортит сюрприз.

Наконец, сохраните модуль для измерения времени в файле, чтобы обращаться к нему в будущем – мы переделаем его с целью измерения производительности альтернативных числовых операций по вычислению квадратного корня в *упражнении*, приведенном в конце настоящей главы. Если вы заинтересованы продолжить данную тему, то мы также поэкспериментируем с методиками хронометража для включений словарей и циклов `for` интерактивно в упражнениях.

Измерение времени выполнения итераций и версий Python с помощью модуля `timeit`

В предыдущем разделе использовались любительские функции измерения времени для сравнения скорости выполнения кода. Как там упоминалось, в состав стандартной библиотеки входит модуль по имени `timeit`, который можно применять похожими способами, но он предлагает добавочную гибкость и может лучше защитить клиентов от влияния ряда отличий между платформами.

Как обычно в Python, важно понимать первоосновы, подобные тем, что иллюстрировались в предыдущем разделе. Принятый в Python подход “батарейки в комплекте” означает, что обычно вы также найдете предварительно реализованные варианты, хотя для их надлежащего использования по-прежнему необходимо знать лежащие в основе идеи. В действительности модуль `timeit` является ярким примером – похоже, с ним связана настоящая история неправильной эксплуатации людьми, которые недостаточно хорошо понимали заключенные в нем принципы. Тем не менее, изучив основы, давайте перейдем к инструменту, который способен автоматизировать большую часть нашей работы.

Базовое использование `timeit`

Прежде чем задействовать модуль `timeit` в более крупных сценариях, начнем с его основ. Вместе с `timeit` тесты указываются посредством либо *вызываемых объектов*, либо *строк операторов*; последние могут хранить множество операторов, если в них применяются разделители ; или символы \n для разрывов строк и пробелы или табуляции для отступа операторов во вложенных блоках (например, \n\t). Тесты могут выполнять действия по настройке и запускаться из *командной строки* и через *вызовы API-интерфейса*, а также из сценариев и в интерактивной подсказке.

Интерактивное использование и вызовы API-интерфейса

Скажем, вызов `repeat` модуля `timeit` возвращает список, дающий суммарное время, которое заняло выполнение теста `number` раз для каждого из прогонов `repeat`. Вызов `min` с таким списком позволяет получить лучшее время среди прогонов и помогает отфильтровать колебания загрузки системы, которые иначе могут искусственно исказить результаты измерения времени в большую сторону.

В следующем коде вызов `repeat` демонстрируется в действии, хронометрируя списковое включение в двух версиях *CPython* и описанной в главе 2 оптимизированной

реализации PyPy (поддерживает код Python 2.7). Результаты дают лучшее суммарное время в секундах из 5 запусков, каждый из которых выполняет строку кода 1000 раз; сама строка кода всякий раз создает 1000-элементный список целых чисел (применимый для разнообразия в первых двух командах запускающий модуль Windows описан в приложении Б второго тома):

```
c:\code> py -3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit...
>>> import timeit
>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.62164939799999956

c:\code> py -2
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] on win32
>>> import timeit
>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.0708020004193198

c:\code> c:\pypy\pypy-1.9\pypy.exe
Python 2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit] on win32
>>> import timeit
>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.0059330329674303905
```

Вы заметите, что PyPy выполняется более чем в 10 раз быстрее CPython 2.7 и более чем в 100 раз быстрее CPython 3.7. Разумеется, это небольшое искусственное оценочное испытание, но выглядит оно опшеломляюще и отражает относительное ранжирование по скорости, которое обычно поддерживается другими тестами, прогоняемыми в данной книге (хотя, как мы увидим, CPython все еще побеждает PyPy в ряде типов кода).

Приведенный тест измеряет скорость спискового включения и целочисленной математики. Последняя варьируется между линейками: CPython 3.X имеет единственный целый тип, а CPython 2.X – короткие и длинные целые, что может объяснить часть величины разницы, но результаты, несомненно, правомерны. Нецелочисленные тесты выдают похожее ранжирование (например, тест с плавающей точкой в решениях упражнений текущей части), и целочисленная математика имеет значение – ускорение на один и два порядка (степень 10) будут претворены в жизнь многими реальными программами, поскольку целые числа и итерации в коде Python вездесущи.

Полученные результаты также отличаются от относительных скоростей разных версий в предыдущем разделе, где CPython 2.7 был немного быстрее CPython 3.7, а PyPy в целом в 10 раз быстрее, что также подтверждается большинством других тестов в книге. Помимо отличающегося типа хронометрируемого здесь кода влияние может оказывать и другая кодовая структура внутри `timeit` – для тестируемых строк модуль `timeit` строит, компилирует и выполняет строку с оператором `def` функции, в который вставлена строка, посредством чего устраняется вызов функции для внутреннего цикла. Однако, как будет показано в следующем разделе, с точки зрения относительной скорости это выглядит несущественным.

Использование в командной строке

В модуле `timeit` предусмотрены обоснованные стандартные параметры. Его можно запускать как сценариев, либо явно указывая имя файла, либо полагаясь на автоматическое нахождение в пути поиска модулей через флаг `-m` интерпретатора Python

(см. приложение А второго тома). Все приведенные ниже команды запускают Python (он же CPython) 3.7. В этом режиме `timeit` сообщает среднее время для *одиночного* цикла, количество которых задано посредством флага `-n`, в микросекундах (`usec`), миллисекундах (`msec`) или секундах (`sec`). Чтобы сравнить полученные здесь результаты со значениями суммарного времени, сообщаемыми другими тестами, их понадобится умножить на количество выполненных циклов — около 500 микросекунд * 1000 циклов дает 500 миллисекунд, или суммарное время полсекунды:

```
c:\code> C:\python37\Lib\timeit.py -n 1000 "[x ** 2 for x in range(1000)]"
1000 loops, best of 3: 506 usec per loop
1000 циклов, лучшее время из 3: 506 микросекунд на цикл

c:\code> python -m timeit -n 1000 "[x ** 2 for x in range(1000)]"
1000 loops, best of 3: 504 usec per loop
1000 циклов, лучшее время из 3: 504 микросекунд на цикл

c:\code> py -3 -m timeit -n 1000 -r 5 "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 505 usec per loop
1000 циклов, лучшее время из 5: 505 микросекунд на цикл
```

В качестве примера мы можем применить командные строки, удостоверившись в том, что выбор вызова таймера не влияет на сравнения скорости между версиями, которые выполнялись в главе до сих пор. Начиная с Python 3.3, по умолчанию используются новые вызовы, что может иметь значение, если точность таймера отличается в широких пределах. Чтобы доказать, что это несущественно, в следующих командах применяется флаг `-c`, который заставляет `timeit` использовать `time.clock` во всех версиях — вариант, помеченный в руководствах по Python 3.3 как устаревший, но он требуется для уравновешивания с предшествующими версиями (ради краткости команд в переменную среды PATH включен путь к PyPy):

```
c:\code> set PATH=%PATH%;C:\pypy\pypy-1.9
c:\code> py -3 -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 502 usec per loop
c:\code> py -2 -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 70.6 usec per loop
c:\code> pypy -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 5.44 usec per loop

C:\code> py -3 -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"
1000 loops, best of 5: 815 usec per loop
C:\code> py -2 -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"
1000 loops, best of 5: 700 usec per loop
C:\code> pypy -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"
1000 loops, best of 5: 61.7 usec per loop
```

По сути, результаты соответствуют проведенным ранее тестам для кода такого типа. В случае применения `x ** 2` версии CPython 2.7 и PyPy оказываются в 10 и 100 раз быстрее CPython 3.7, что говорит о неважности выбора таймера. Для функции `abs(x)`, время выполнения которой ранее измерялось с помощью любительского таймера (`timeseqs.py`), версии CPython 2.7 и PyPy работают быстрее Python 3.7 соответственно на небольшое константное значение и в 10 раз. Отсюда вытекает, что отличающаяся кодовая структура `timeit` не влияет на относительные сравнения — величина разницы в скоростях целиком определяется типом тестируемого кода.

Есть один тонкий момент: обратите внимание, что результаты последних трех тестов, имитирующие тесты, которые ранее запускались для любительского таймера, по существу такие же, но несут в себе небольшие совокупные издержки. Такие издержки

обусловлены отличиями в использовании функции `range` – раньше она возвращала созданный список, но здесь мы имеем дело либо с генератором Python 3.X, либо со списком Python 2.X, который заново строится на каждом внутреннем цикле. Другими словами, мы измеряем время не в точности того же самого кода, но относительные скорости проверяемых версий Python сохраняются.

Измерение времени выполнения многострочных операторов

Для измерения времени более крупных многострочных порций кода в режиме вызовов API-интерфейса необходимо применять разрывы строк и табуляции или пробелы с целью удовлетворения требованиям синтаксиса Python; код, прочитанный из файла исходного кода, уже им удовлетворяет. Поскольку в этом режиме вы передаете строковые объекты Python функции `PyFunction`, то нет никаких соображений касательно командной оболочки, но нужно отменять вложенные кавычки, если они встречаются. Скажем, следующий код измеряет время альтернативных версий циклов из главы 13; ту же самую схему можно использовать при измерении времени выполнения альтернативных версий для чтения строк из файла, представленных в главе 14:

```
c:\code> py -3
>>> import timeit
>>> min(timeit.repeat(number=10000, repeat=3,
    stmt="L = [1, 2, 3, 4, 5]\nfor i in range(len(L)):\n    L[i] += 1"))
0.01397292797131814

>>> min(timeit.repeat(number=10000, repeat=3,
    stmt="L = [1, 2, 3, 4, 5]\ni=0\nwhile i < len(L):\n    L[i] += 1\n    i += 1"))
0.015452276471516813

>>> min(timeit.repeat(number=10000, repeat=3,
    stmt="L = [1, 2, 3, 4, 5]\nM = [x + 1 for x in L]"))
0.009464995838568635
```

Чтобы работать с такими многострочными операторами в режиме *командной строки*, понадобится передавать каждую строку оператора как отдельный аргумент с пребельными символами для отступа – модуль `timeit` объединяет все строки вместе с символами новой строки между ними, и позже делает отступы в собственных целях вложения операторов. В этом режиме ведущие пробелы могут работать для отступов лучше табуляций, и аргументы кода важно помещать в кавычки, если того требует командная оболочка:

```
c:\code> py -3 -m timeit -n 1000 -r 3 "L = [1,2,3,4,5]" "i=0" "while i < len(L) :"
    "    L[i] += 1" "    i += 1"
1000 loops, best of 3: 1.54 usec per loop

c:\code> py -3 -m timeit -n 1000 -r 3 "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
1000 loops, best of 3: 0.959 usec per loop
```

Другие режимы использования: настройка, суммарное время и объекты

Модуль `timeit` также позволяет предоставлять код *настройки*, который выполняется в области видимости главного оператора, но время его выполнения не учитывается в суммарном времени главного оператора. Это потенциально удобно для кода инициализации, импортирующего обязательные модули, определяющего тестируемую функцию и создающего тестовые данные, время выполнения которого желательно исключить из суммарного времени. Поскольку код настройки запускается в той же области видимости, все создаваемые им имена доступны главному оператору; имена, определяемые в интерактивной оболочке, обычно недоступны.

Для указания кода настройки в режиме командной строки применяется строка после флага `-s` (или несколько в случае многострочного кода настройки), а в режиме вызовов API-интерфейса — аргумент `setup`. В итоге тесты можно ориентировать более четко, как показано ниже, где инициализация списка выносится в код настройки, чтобы измерять время только итерации. Но запомните в качестве эмпирического правила, что чем больше кода вы включаете в тестируемый оператор, тем более применимыми обычно будут результаты тестирования к реалистичному коду:

```
c:\code> python -m timeit -n 1000 -r 3 "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
1000 loops, best of 3: 0.956 usec per loop
c:\code> python -m timeit -n 1000 -r 3 -s "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
1000 loops, best of 3: 0.775 usec per loop
```

Ниже демонстрируется настройка в режиме API-вызовов. Здесь измеряется время выполнения варианта, основанного на сортировке, из примера нахождения минимального значения, который рассматривался в главе 18 — упорядоченные диапазоны сортируются гораздо быстрее случайных чисел, и они сортируются быстрее, чем линейно просматриваются в коде примера в Python 3.7 (расположенные рядом строки объединяются):

```
>>> from timeit import repeat
>>> min(repeat(number=1000, repeat=3,
    setup='from mins import min1, min2, min3\n'
    'vals=list(range(1000))',
    stmt= 'min3(*vals)'),)
0.017538969001179794
>>> min(repeat(number=1000, repeat=3,
    setup='from mins import min1, min2, min3\n'
    'import random\nvals=[random.random() for i in range(1000)]',
    stmt= 'min3(*vals)'),)
0.13474658399900363
```

С помощью модуля `timeit` вы можете запрашивать только суммарное время, использовать API-интерфейс класса модуля, применятьываемые объекты времени вместо строк, принимать автоматические счетчики циклов, а также использовать методики на основе классов, дополнительные переключатели командной строки и варианты аргументов API-интерфейса, описать которые в книге попросту не хватит места — детали ищите в руководстве по библиотеке Python:

```
c:\code> py -3
>>> import timeit
>>> timeit.timeit(stmt='[x ** 2 for x in range(1000)]', number=1000)
# Суммарное время
0.4288552190000008
>>> timeit.Timer(stmt='[x ** 2 for x in range(1000)]').timeit(1000)
# API-интерфейс класса модуля
0.4739268729999999
>>> timeit.repeat(stmt='[x ** 2 for x in range(1000)]', number=1000, repeat=3)
[0.43331713899999613, 0.4261321769999995, 0.42656452599999994]
>>> def testcase():
    y = [x ** 2 for x in range(1000)] # Вызываемые объекты или строки кода
>>> min(timeit.repeat(stmt=testcase, number=1000, repeat=3))
0.43041660699999795
```

Модуль и сценарий оценочных испытаний: `timeit`

Вместо того чтобы вдаваться в дополнительные подробности касательно модуля `timeit`, давайте изучим программу, которая его применяет для измерения времени кодовых альтернатив и версий Python. Файл `pybench.py` с показанным ниже содержимым настроен на измерение времени набора операторов в сценариях, которые импортируют и используют данный файл под управлением либо версии, выполняющей его, либо всех версий Python, перечисленных в списке. Он задействует ряд описываемых далее инструментов уровня приложения. Тем не менее, поскольку в нем по большей части применяются идеи, которые уже исследованы и щедро документированы, я склонен отнести его к категории материалов для самостоятельного изучения и упражнений по чтению кода Python.

```
"""
pybench.py: тестирует скорость одной и более версий Python
на наборе простых эталонных тестов в виде строк кода.
Функция runner допускает разнообразные операторы в stmts.
Сама система выполняется в Python 2.X и 3.X и может
порождать вывод в обеих линейках.

Использует модуль timeit для тестирования либо версии Python,
выполняющей этот сценарий, через вызовы API-интерфейса, либо
набора версий Python за счет чтения порожденного вывода командной
строки (os.popen) посредством флага -m интерпретатора Python,
позволяющего находить timeit в пути поиска модулей.

Заменяет $listif3 на list в генераторах для Python 3.X и на пустую строку
для Python 2.X, заставляя Python 3.X делать ту же самую работу, что и Python 2.X.
В режиме командной строки многострочные операторы должны разделяться на
отдельные
аргументы в кавычках для каждой строки и все символы \t в отступах должны
заменяться четырьмя пробелами ради единобразия.

Предостережения: режим командной строки (только) может потерпеть неудачу, если
внутри оператора содержатся двойные кавычки, строка оператора в кавычках вообще
несовместима с командной оболочкой или командная строка превышает предел длины
в командной оболочке платформы -- применяйте режим вызовов API-интерфейса
или любительский таймер; пока не поддерживает оператор настройки: в том виде,
как есть, время всех операторов в stmts учитывается в суммарном времени.
"""

import sys, os, timeit
defnum, defrep= 1000, 5          # Может варьироваться от оператора к оператору
def runner(stmts, pythons=None, tracecmd=False):
    """
    Основная логика: запускать тесты на входных списках, режимами использования
    управляет вызывающий код.
    stmts: [(количество?, повторений?, строка-оператора)], заменяет $listif3
    в строке оператора
    pythons: None = только эта версия Python или [(версия-Python-3?,,
    путь-к-исполняемому-файлу-Python)]
    """
    print(sys.version)
    for (number, repeat, stmt) in stmts:
        number = number or defnum
        repeat = repeat or defrep      # 0 = стандартное значение
```

```

if not pythons:
    # Запустить оператор stmt в этой версии Python: вызов API-интерфейса
    # Нет необходимости разделять строки или помещать в кавычки
    ispy3 = sys.version[0] == '3'
    stmt = stmt.replace('$listif3', 'list' if ispy3 else '')
    best = min(timeit.repeat(stmt=stmt, number=number, repeat=repeat))
    print('%.4f [%r]' % (best, stmt[:70]))
else:
    # Запустить оператор stmt во всех версиях Python: командная строка
    # Разделить строки на аргументы в кавычках
    print('-' * 80)
    print('[%r]' % stmt)
    for (ispy3, python) in pythons:
        stmt1 = stmt.replace('$listif3', 'list' if ispy3 else '')
        stmt1 = stmt1.replace('\t', ' ' * 4)
        lines = stmt1.split('\n')
        args = ' '.join(['"%s"' % line for line in lines])
        cmd = '%s -m timeit -n %s -r %s %s' % (python, number, repeat, args)
        print(python)
        if tracecmd: print(cmd)
        print('\t' + os.popen(cmd).read().rstrip())

```

Однако данный файл отражает лишь половину картины. Тестовые сценарии применяют функцию модуля, передавая конкретные (хотя и изменяемые) списки операторов и подлежащих тестированию версий Python в зависимости от желаемого режима использования. Скажем, сценарий `pybench_cases.py`, содержимое которого приведено далее, тестирует несколько операторов и версий Python, позволяя с помощью аргументов командной строки определять часть своей работы. Указание `-a` означает тестирование всех перечисленных версий Python, а не только одной, наличие `-t` заставляет выполнять трассировку создаваемых командных строк, чтобы можно было видеть, каким образом многострочные операторы и отступы обрабатываются согласно показанным ранее форматам (детали описаны в строках документации обоих файлов):

```

"""
pybench_cases.py: запускает pybench на наборе версий Python и операторов.
Выбирайте режимы путем редактирования этого сценария либо использования
аргументов командной строки (в sys.argv): например, запускайте
C:\python27\python pybench_cases.py, чтобы протестировать только одну
версию Python из перечисленных в stmts, pybench_cases.py -a
для тестирования всех версий Python и py -3 pybench_cases.py -a -t
для трассировки командных строк.
"""

import pybench, sys
pythons = [                                         # (Python 3?, путь)
    (1, 'C:\python33\python'),                      # Итерации
    (0, 'C:\python27\python'),
    (0, 'C:\pypy\pypy-1.9\pypy')
]
stmts = [                                         # (количество, повторения, оператор)
    (0, 0, "[x ** 2 for x in range(1000)]"),       # Итерации
    (0, 0, "res=[]\nfor x in range(1000): res.append(x ** 2)"), # \n = несколько
                                                       # операторов
    (0, 0, "$listif3(map(lambda x: x ** 2, range(1000))))", # \n\t = отступ

```

```

(0, 0, "list(x ** 2 for x in range(1000))"),           # $ = list или ''
(0, 0, "s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"),
    # Строковые операции
(0, 0, "s = '?'\nfor i in range(10000): s += '?'"),
]

traceecmd = '-t' in sys.argv      # -t: трассировать командные строки?
pythons = pythons if '-a' in sys.argv else None      # -a: все версии Python
                                                # в списке, иначе одну?

pybench.runner(stmts, pythons, traceecmd)

```

Результаты запуска сценария оценочных испытаний

Ниже приведен вывод сценария при его запуске для тестирования *специфической версии* Python (которая выполняет сценарий) – в таком режиме применяются прямые вызовы API-интерфейса, а не командные строки, с выводом суммарного времени слева и тестируемым оператором справа. В первых двух тестах снова используется запускающий модуль Windows для измерения времени *CPython* 3.7 и 2.7, а в третьем – выпуск 1.9 реализации *PyPy*:

```

c:\code> py -3 pybench_cases.py
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
0.4270  ['[x ** 2 for x in range(1000)]']
0.5445  ['res=[]\nfor x in range(1000): res.append(x ** 2)']
0.5073  ['list(map(lambda x: x ** 2, range(1000)))']
0.4792  ['list(x ** 2 for x in range(1000))']
1.3119  ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
2.4669  ["s = '?'\nfor i in range(10000): s += '?'"]

c:\code> py -2 pybench_cases.py
2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)]
0.0696  ['[x ** 2 for x in range(1000)]']
0.1285  ['res=[]\nfor x in range(1000): res.append(x ** 2)']
0.1636  ['(map(lambda x: x ** 2, range(1000)))']
0.0952  ['list(x ** 2 for x in range(1000))']
0.6143  ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
2.0657  ["s = '?'\nfor i in range(10000): s += '?'"]

c:\code> c:\pypy\pypy-1.9\pypy pybench_cases.py
2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit]
0.0059  ['[x ** 2 for x in range(1000)]']
0.0102  ['res=[]\nfor x in range(1000): res.append(x ** 2)']
0.0099  ['(map(lambda x: x ** 2, range(1000)))']
0.0156  ['list(x ** 2 for x in range(1000))']
0.1298  ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
5.5242  ["s = '?'\nfor i in range(10000): s += '?'"]

```

Далее показан вывод сценария при его запуске для тестирования *множества версий Python* на каждой строке оператора. В таком режиме сам сценарий выполняется под управлением версии Python 3.7, но выдает командные строки оболочки, которые запускают другие версии Python, чтобы выполнить модуль *timeit* на тестовых строках операторов. В этом режиме требуется разделение, форматирование и помещение в кавычки многострочных операторов для применения в командных строках согласно ожиданиям модуля *timeit* и требованиям командной оболочки.

Данный режим полагается на флаг *-t* командной строки интерпретатора Python для нахождения *timeit* в пути поиска модулей и его выполнения как сценария, а также

на стандартные библиотечные инструменты `os.popen` и `sys.argv` для запуска команды оболочки и инспектирования аргументов командной строки. Дополнительные сведения о них ищите в руководствах по Python и других источниках; инструмент `os.popen` использовался при рассмотрении файлов в главе 9 и циклов в главе 13. Флаг `-t` позволяет следить за выполнением командных строк:

```
c:\code> py -3 pybench_cases.py -a
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
-----
['[x ** 2 for x in range(1000)]']
C:\python37\python
    1000 loops, best of 5: 487 usec per loop
C:\python27\python
    1000 loops, best of 5: 71.4 usec per loop
C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 5.71 usec per loop
-----
['res=[]\nfor x in range(1000): res.append(x ** 2)']
C:\python37\python
    1000 loops, best of 5: 558 usec per loop
C:\python27\python
    1000 loops, best of 5: 130 usec per loop
C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 9.81 usec per loop
-----
 ['$listif3(map(lambda x: x ** 2, range(1000)))']
C:\python37\python
    1000 loops, best of 5: 589 usec per loop
C:\python27\python
    1000 loops, best of 5: 161 usec per loop
C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 9.45 usec per loop
-----
['list(x ** 2 for x in range(1000))']
C:\python37\python
    1000 loops, best of 5: 553 usec per loop
C:\python27\python
    1000 loops, best of 5: 92.3 usec per loop
C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 15.1 usec per loop
-----
["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
C:\python37\python
    1000 loops, best of 5: 848 usec per loop
C:\python27\python
    1000 loops, best of 5: 614 usec per loop
C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 118 usec per loop
-----
["s = ?'\nfor i in range(10000): s += '?'"]
C:\python37\python
    1000 loops, best of 5: 2.83 msec per loop
C:\python27\python
    1000 loops, best of 5: 1.94 msec per loop
C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 5.68 msec per loop
```

Как видите, в большинстве тестов версия CPython 2.7 по-прежнему быстрее CPython 3.7, а PyPy значительно быстрее обеих версий за исключением последнего теста, где PyPy в два раза медленнее CPython предположительно из-за отличий в управлении памятью. С другой стороны, результаты измерения времени часто в лучшем случае относительны. Помимо упомянутых ранее предупреждений общего характера, касающихся измерения времени, ниже перечислены дополнительные замечания.

- Модуль `timeit` может искажать результаты по причинам, которые выходят за рамки исследованных здесь (скажем, из-за сборки мусора).
- Существует базовый уровень издержек, отличающийся в зависимости от версии Python, который здесь игнорируется (но выглядит тривиальным).
- Сценарий выполняет очень маленькие операторы, которые могут отражать или не отражать реальный код (но все равно они действительны).
- Результаты могут изредка варьироваться по причинам, которые кажутся случайными (здесь может помочь применение времени процесса).
- Все результаты крайне подвержены изменениям с течением времени (на самом деле в каждом новом выпуске Python!).

Таким образом, вы должны выработать собственные заключения в отношении продемонстрированных выше показаний и прогонять такие тесты на своих версиях Python и компьютерах для получения более значимых результатов. Для хронометража базового уровня издержек каждой версии Python запустите `timeit`, не передавая никаких операторов в аргументе, или передайте оператор `pass`, что эквивалентно.

Продолжаем забавляться с оценочными испытаниями

Чтобы еще лучше уловить суть, попробуйте выполнить сценарий в других версиях Python и с другими строками операторов. В файле `pybench_cases2.py` из пакета примеров добавлены тесты для сравнения CPython 3.7 и CPython 3.2, бета-версии PyPy 2.0 и PyPy 1.9, а также демонстрации дополнительных вариантов использования.

Выигрыш `map` и редкий проигрыш PyPy

Например, следующие тесты в `pybench_cases2.py` измеряют влияние нагрузки со стороны других итерационных операций с вызовом функции, что повышает шансы на выигрыш у `map`; согласно замечанию, сделанному ранее в главе, `map` обычно проигрывает в целом из-за своей ассоциации с вызовами функций:

```
# Файл pybench_cases2.py
pythons += [
    (1, 'C:\python32\python'),
    (0, 'C:\pypy\pypy-2.0-beta1\pypy')]
stmts += [
    # Использовать вызовы функций: map выигрывает
    (0, 0, "[ord(x) for x in 'spam' * 2500]"),
    (0, 0, "res=[]\nfor x in 'spam' * 2500: res.append(ord(x))"),
    (0, 0, "$listif3(map(ord, 'spam' * 2500)))"),
    (0, 0, "list(ord(x) for x in 'spam' * 2500)"),
    # Множества и словари
    (0, 0, "{x ** 2 for x in range(1000)}"),
    (0, 0, "s=set()\nfor x in range(1000): s.add(x ** 2)")]
```

```

(0, 0, "{x: x ** 2 for x in range(1000)}"),
(0, 0, "d={}\\nfor x in range(1000): d[x] = x ** 2"),
# Патологический случай: 301030 цифр
(1, 1, "len(str(2**1000000)))]) # На этот раз PyPy проигрывает

```

По результатам выполнения сценария в CPython 3.X видно, что `map` оказывается быстрее, когда вызовы функций уравнивают шансы (ранее функция `map` проигрывала в других тестах, выполнивших операцию `x ** 2`):

```

c:\code> py -3 pybench_cases2.py
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
0.7237 "[ord(x) for x in 'spam' * 2500]"
1.3471 ["res=[]\\nfor x in 'spam' * 2500: res.append(ord(x))"]
0.6160 ["list(map(ord, 'spam' * 2500))"]
1.1244 ["list(ord(x) for x in 'spam' * 2500)"]
0.5446 ['{x ** 2 for x in range(1000)}']
0.6053 ['s=set()\\nfor x in range(1000): s.add(x ** 2)']
0.5278 [''{x: x ** 2 for x in range(1000)}'']
0.5414 ['d={}\\nfor x in range(1000): d[x] = x ** 2'']
1.8933 ['len(str(2**1000000))']

```

Как и прежде, в текущее время Python 2.X работает быстрее Python 3.X, а PyPy быстрее на всех тестах кроме последнего, по скорости выполнения которого он проигрывает на порядок (в 10 раз), хотя при выполнении всех остальных тестов в такой же степени выигрывает. Тем не менее, если вы запустите файловые тесты, написанные в `pybench_cases2.py`, то заметите, что PyPy также проигрывает CPython, когда читает файлы строка за строкой, как в случае следующего тестового кортежа в списке `stmts`:

```
(0, 0, "f=open('C:/Python37/Lib/pdb.py')\\nfor line in f: x=line\\nf.close()",
```

Тест открывает и читает 1 728-строчный текстовый файл размером около 62 Кбайт строка за строкой с применением файловых итераторов. В этом тесте CPython 2.7 в два раза быстрее CPython 3.7, но PyPy снова на порядок медленнее, чем CPython в целом. Данный случай можно найти в файлах результатов `pybench_cases2` или проверить интерактивно либо в командной строке (что внутренне делает `pybench`):

```

c:\code> py -3 -m timeit -n 1000 -r 5 "f=open('C:/Python37/Lib/pdb.py')"
"for line in f: x=line" "f.close()"

>>> import timeit
>>> min(timeit.repeat(number=1000, repeat=5,
stmt="f=open('C:/Python37/Lib/pdb.py')\\nfor line in f: x=line\\nf.close()")

```

Еще один пример измерения скорости выполнения списковых включений и чтения файлов PyPy приведен в файле `listcomp-speed.txt` из пакета примеров; в нем используются прямые командные строки PyPy для запуска кода из главы 14 с похожими результатами: построчный ввод PyPy медленнее приблизительно на порядок.

Вывод из других версий Python здесь не показан ради экономии места, а также потому, что на момент чтения вами книги ситуация наверняка изменится. Как обычно, разные типы кода могут демонстрировать разные типы производительности. Наряду с тем, что реализация PyPy способна оптимизировать большинство алгоритмического кода, вас самих она может как оптимизировать, так и нет. Вы найдете дополнительные результаты в пакете примеров книги, но может быть лучше самостоятельно проверить эти сведения сейчас, чем получить, возможно, другие результаты в будущем.

Снова о влиянии вызовов функций

Как предполагалось ранее, тар также выигрывает для добавленных функций, определяемых пользователем; следующие тесты подтверждают сделанное ранее заявление о том, что тар выигрывает состязание у CPython, если в альтернативных версиях должна применяться любая функция:

```
stmts = [
    (0, 0, "def f(x): return x\n[f(x) for x in 'spam' * 2500]"),
    (0, 0, "def f(x): return x\n[res=[]\nfor x in 'spam' * 2500: res.append(f(x))"],
    (0, 0, "def f(x): return x\n$listify3(map(f, 'spam' * 2500))"),
    (0, 0, "def f(x): return x\nlist(f(x) for x in 'spam' * 2500))"]

c:\code> py -3 pybench_cases2.py
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
1.5400 ["def f(x): return x\n[f(x) for x in 'spam' * 2500]"]
2.0506 ["def f(x): return x\nres=[]\nfor x in 'spam' * 2500: res.
append(f(x))"]
1.2489 ["def f(x): return x\nlist(map(f, 'spam' * 2500))"]
1.6526 ["def f(x): return x\nlist(f(x) for x in 'spam' * 2500)"]
```

Сравните это с тестами `ord` из предыдущего раздела; хотя определяемые пользователем функции могут работать медленнее встроенных, крупный скачок скорости в наши дни, похоже, коснулся функций в целом, будь они встроенными или нет. Обратите внимание, что суммарное время здесь включает затраты на создание вспомогательной функции, правда, только один раз на 10 000 повторений внутреннего цикла — пренебрежимо малый фактор как с точки зрения здравого смысла, так и для запуска дополнительных тестов.

Сравнение методик: любительская или батарейки

На перспективу давайте посмотрим, как полученные в данном разделе результаты, основанные на модуле `timeit`, соотносятся с результатами, которые получены с помощью любительского таймера в предыдущем разделе, запустив файл `timeseqs3.py` из пакета примеров — в нем используется любительский таймер, но выполняется та же самая операция `x ** 2` с применением таких же счетчиков повторений, как в `pybench_cases.py`:

```
c:\code> py -3 timeseqs3.py
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
forLoop : 0.50858 => [0...998001]
listComp : 0.40494 => [0...998001]
mapCall : 0.49577 => [0...998001]
genExpr : 0.46281 => [0...998001]
genFunc : 0.46362 => [0...998001]

c:\code> py -3 pybench_cases.py
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
0.4294 ['[x ** 2 for x in range(1000)]']
0.5434 ['res=[]\nfor x in range(1000): res.append(x ** 2)']
0.5096 ['list(map(lambda x: x ** 2, range(1000)))']
0.4820 ['list(x ** 2 for x in range(1000))']
1.3042 ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
2.4690 ["s = '?'`\nfor i in range(10000): s += '?'"]
```

Результаты, полученные с использованием любительского таймера, очень похожи на результаты, основанные на `pybench`, которые получались в этом разделе с помощью `timeit`, хотя и не совсем так, как яблоко на яблоко. В файле `timeseqs3.py`,

основанном на любительском таймере, не только производится вызов функции для цикла измерения суммарного времени и привносятся небольшие издержки со стороны логики нахождения лучшего времени самого таймера, но в его внутреннем цикле также применяется готовый список, а не генератор `range` из Python 3.X. Видимо это в итоге делает его чуть быстрее на сопоставимых тестах (я бы назвал данный пример “контролем корректности”, но не уверен, что такой термин применим при оценочных испытаниях).

Простор для улучшений: настройка

Подобно большинству программного обеспечения программа в настоящем разделе открыта и может произвольно расширяться. Как один пример, в файлах `pybench2.py` и `pybench2_cases.py` внутри пакета примеров добавлена поддержка описанной ранее возможности указания операторов *настройки* для модуля `timeit` в режимах API-вызовов и командной строки.

Первоначально эта возможность была опущена ради краткости и, откровенно говоря, из-за того, что тестам она, похоже, не требовалась – измерение времени выполнения кода большого объема дает более законченное представление при сравнении версий Python, а действия по настройке влекут за собой одинаковые затраты, когда проводится хронометраж альтернатив в одной версии Python. Даже при этих условиях иногда полезно предоставлять код настройки, который выполняется однократно в области видимости тестируемого кода, но затраченное время не учитывается в суммарном времени – скажем, импортирование модулей, инициализация объектов или определение функций.

Полностью содержимое указанных двух файлов приводиться не будет, но в качестве примера развития программного обеспечения ниже показаны их важные варварирующиеся части. Что касается тестируемого оператора, то оператор кода настройки передается в том виде, как есть, в режиме вызовов API-интерфейса, но он разделяется и снабжается отступами в режиме командной строки с передачей по одной строке в аргументе `-s` (`$listif3` не используется, т.к. код настройки не хронометрируется):

```
# Файл pybench2.py
...
def runner(stmts, pythons=None, tracecmd=False):
    for (number, repeat, setup, stmt) in stmts:
        if not pythons:
            ...
            best = min(timeit.repeat(
                setup=setup, stmt=stmt, number=number, repeat=repeat))
        else:
            setup = setup.replace('\t', ' ' * 4)
            setup = '.join('-s "%s" % line for line in setup.split(\n))'
            ...
            for (ispy3, python) in pythons:
                ...
                cmd = '%s -m timeit -n %s -r %s %s %s' %
                    (python, number, repeat, setup, args)

# Файл pybench2_cases.py
import pybench2, sys
...
stmts = [ # (num,rpt,setup,stmt)
    (0, 0, "", "[x ** 2 for x in range(1000)]"),
    (0, 0, "", "res=[]\nfor x in range(1000): res.append(x ** 2)"),
    ...]
```

```

(0, 0, "def f(x):\n\treturn x",
 "[f(x) for x in 'spam' * 2500]"),
(0, 0, "def f(x):\n\treturn x",
 "res=[]\nfor x in 'spam' * 2500:\n\tres.append(f(x))"),
(0, 0, "L = [1, 2, 3, 4, 5]", "for i in range(len(L)):\n\tL[i] += 1"),
(0, 0, "L = [1, 2, 3, 4, 5]", "i=0\nwhile i < len(L):\n\tL[i] += 1\n\ti += 1")
...
pybench2.runner(stmts, pythons, tracecmd)

```

Запустите этот сценарий с флагами командной строки `-a` и `-t`, чтобы посмотреть, каким образом конструируются командные строки для кода настройки. Например, следующий кортеж спецификаций теста генерирует командную строку для Python 3.7 – возможно, не особо приглядную внешне, но достаточную, чтобы передать строки из Windows в `timeit`, которые объединяются с разрывами строк и вставляются внутри генерированной функции измерения времени с подходящими отступами:

```

(0, 0, "def f(x):\n\treturn x",
 "res=[]\nfor x in 'spam' * 2500:\n\tres.append(f(x))")
C:\python37\python -m timeit -n 1000 -r 5 -s "def f(x):" -s "    return x" "res=[]"
"for x in 'spam' * 2500:" "    res.append(f(x))"

```

В режиме вызовов API-интерфейса строки с кодом передаются в неименном виде, поскольку нет необходимости умиротворять командную оболочку, так что встроенных табуляций и символов конца строки вполне достаточно. Поэкспериментируйте самостоятельно, чтобы раскрыть для себя еще больше аспектов, касающихся скорости выполнения кодовых альтернатив Python. В итоге вы можете столкнуться с ограничениями оболочки для более крупных разделов кода в режиме командной строки, но любительский таймер и основанный на `timeit` режим вызовов API-интерфейса модуля `pybench` поддерживают более произвольный код. Оценочные испытания могут оказаться самым приятным времяпрепровождением, так что дальнейшие усовершенствования оставляются в качестве упражнений.

Другие темы, связанные с оценочными испытаниями: тест `pystone`

Главное внимание в главе уделяется основам хронометража, которые вы можете задействовать в своем коде, которые применимы к оценочным испытаниям Python в общем и которые служат распространенными сценариями использования при разработке более крупных примеров в книге. Однако оценочные испытания Python являются более широкой и насыщенной предметной областью, чем предполагалось до сих пор. Если вас интересуют дальнейшие исследования данной темы, тогда поищите ссылки в веб-сети. Помимо прочего вы найдете:

- `pystone.py` – программа, предназначенная для измерения скорости Python в широком диапазоне кода (<https://github.com/blackberry/Python/blob/master/Python-3/Lib/test/pystone.py>);
- <https://speed.python.org> – сайт проекта для координации работы над распространенными эталонными тестами Python;
- <https://speed.pyru.org> – сайт оценочных испытаний PyRu, который частично эмулирует предыдущий пункт.

Скажем, тест `pystone` основан на программе оценочных испытаний языка C, которая была переведена на Python первоначальным создателем Python Гвидо ван Россумом.

Он предлагает еще один способ для измерения относительных скоростей выполнения реализаций Python и, кажется, в целом поддерживает наши выводы:

```
c:\Python37\Lib\test> cd C:\python37\lib\test
c:\Python37\Lib\test> py -3 pystone.py
Pystone(1.1) time for 50000 passes = 0.685303
This machine benchmarks at 72960.4 pystones/second
Pystone(1.1) время для 50000 проходов = 0.685303
Эталонные тесты на этой машине прошли со скоростью 72960.4 единиц pystone/c

c:\Python37\Lib\test> cd c:\python27\lib\test
c:\Python27\Lib\test> py -2 pystone.py
Pystone(1.1) time for 50000 passes = 0.463547
This machine benchmarks at 107864 pystones/second
Pystone(1.1) время для 50000 проходов = 0.463547
Эталонные тесты на этой машине прошли со скоростью 107864 единиц pystone/c

c:\Python27\Lib\test> c:\pypy\pypy-1.9\pypy pystone.py
Pystone(1.1) time for 50000 passes = 0.099975
This machine benchmarks at 500125 pystones/second
Pystone(1.1) время для 50000 проходов = 0.099975
Эталонные тесты на этой машине прошли со скоростью 500125 единиц pystone/c
```

Поскольку наступило время завершить главу, последнего должно быть достаточно для независимого подтверждения результатов выполненных тестов. Исследование смысла результатов теста `pystone` оставлено в качестве упражнения; его код не идентичен в версиях Python 3.X и 2.X, но похоже отличается лишь в том, что касается операций вывода и инициализации. Также имейте в виду, что оценочные испытания являются всего лишь одним из многих аспектов анализа кода Python; подсказки о вариантах в связанных предметных областях (например, тестировании) ищите в обзоре инструментов разработки на Python в главе 36.

Затруднения, связанные с функциями

Теперь, когда мы добрались до конца истории о функциях, давайте проанализируем связанные с ними распространенные затруднения. Функции обладают рядом тонких особенностей, о существовании которых вы можете даже не подозревать. Все они относительно малоизвестны, а некоторые полностью ушли в сторону от языка в последних выпусках, но большинство их продолжают сбивать с толку новичков.

Локальные имена распознаются статически

Как вам уже известно, по умолчанию Python классифицирует имена, присваиваемые в функции, как *локальные*; они существуют в области видимости функции только в период, пока она выполняется. Но вы можете не осознавать тот факт, что Python распознает локальные имена статически, когда компилирует код `def`, а не во время встречи присваиваний именам на стадии выполнения. Это приводит к одной из наиболее распространенных странностей, публикуемых в новостных группах начинающими. Обычно имя, которое не было присвоено в функции, ищется во включающем модуле:

```
>>> X = 99
>>> def selector():
...     print(X)
...             # Имя X используется, но не присваивается
...             # Имя X найдено в глобальной области видимости
>>> selector()
99
```

Здесь имя X в функции распознается как имя X в модуле. Но посмотрим, что произойдет, если добавить присваивание X после ссылки на него:

```
>>> def selector():
    print(X)          # Пока еще не существует!
    X = 88           # X классифицируется как локальное имя (повсюду)
                     # Может также произойти в случае import X, def X и т.д.
>>> selector()
UnboundLocalError: local variable 'X' referenced before assignment
Ошибка несвязанной локальной переменной: ссылка на локальную переменную X
перед ее присваиванием
```

Выдается сообщение об ошибочном применении имени, но причина ее довольно тонкая. Python читает и компилирует этот код, когда он набирается в интерактивной подсказке либо импортируется из модуля. На стадии компиляции Python видит присваивание X и решает, что X будет локальным именем повсюду в функции. Но когда функция фактически запущена, из-за того, что к моменту выполнения print присваивание еще не произошло, Python сообщает о попытке использования неопределенного имени. Согласно правилам применения имен он обязан выдать такое сообщение – локальное имя X используется до его присваивания. В действительности любое присваивание в теле функции делает имя локальным. Такое поведение касается импортирования, операций =, вложенных def, вложенных классов и т.д.

Проблема обусловлена тем, что присвоенные имена трактуются как локальные повсюду в функции, а не только после операторов, где они присваивались. На самом деле предыдущий пример неоднозначен: заключалось ли намерение в том, чтобы вывести глобальное имя X и создать локальное имя X, либо просто в коде была допущена ошибка? Поскольку Python рассматривает X как локальное имя везде в функции, ситуация воспринимается как ошибочная; если подразумевается вывод глобального имени X, тогда его необходимо объявить в операторе global:

```
>>> def selector():
    global X      # Сделать X глобальным именем (повсюду)
    print(X)
    X = 88

>>> selector()
99
```

Тем не менее, помните о том, что тогда присваивание будет изменять глобальное имя X, а не локальное X. Внутри функции невозможно применять локальную и глобальную версии того же самого простого имени. Если действительно намерение состояло в том, чтобы вывести глобальную переменную и затем установить локальную переменную с тем же именем, тогда придется импортировать включающий модуль и использовать для обращения к глобальной переменной форму записи атрибута модуля:

```
>>> X = 99
>>> def selector():
    import __main__      # Импортирование включающего модуля
    print(__main__.X)   # Уточнение для обращения к глобальной версии имени
    X = 88             # Имя X без уточнения классифицируется как локальное
    print(X)           # Выводит локальную версию имени

>>> selector()
99
88
```

Уточнение (часть .X) извлекает значение из объекта в пространстве имен. Пространством имен интерактивной подсказки является модуль под названием `main__`, так что `__main__.X` обеспечивает доступ к глобальной версии X. Если суть сказанного не вполне ясна, тогда перечитайте главу 17.

В последних версиях Python положение дел несколько улучшилось за счет выдачи более специфичного сообщения об ошибке типа “несвязанная локальная переменная”, которая была показана ранее (ранее просто выдавалось универсальное сообщение об ошибке, касающейся имени); однако в целом это затруднение все еще встречается.

Стандартные значения и изменяемые объекты

Как кратко отмечалось в главах 17 и 18, изменяемые объекты для аргументов со стандартными значениями способны предохранять состояние между вызовами, хотя это часто не ожидается. В общем случае стандартные значения аргументов оцениваются и сохраняются однократно при выполнении оператора `def`, а не каждый раз, когда результирующая функция вызывается в более позднее время. Внутренне Python хранит по одному объекту для аргумента со стандартным значением, присоединяя его к самой функции.

Обычно именно такое поведение и требуется — поскольку стандартные значения оцениваются во время выполнения `def`, они позволяют при необходимости сохранять значения из объемлющей области видимости (как объясняется далее, функции, определяемые внутри циклов посредством фабрик, могут даже полагаться на такое поведение). Но из-за того, что стандартное значение сохраняется между вызовами, вы должны проявлять осмотрительность при модификации изменяемых объектов, выбранных для стандартных значений. Скажем, в следующей функции в качестве стандартного значения применяется пустой список, который модифицируется на месте каждый раз, когда функция вызывается:

```
>>> def saver(x=[]):      # Немедленно сохраняет объект списка
    x.append(1)           # Каждый раз изменяет тот же самый объект!
    print(x)

>>> saver([2])          # Стандартное значение не используется
[2, 1]
>>> saver()             # Стандартное значение используется
[1]
>>> saver()             # Растет с каждым вызовом!
[1, 1]
>>> saver()
[1, 1, 1]
```

Одни рассматривают такое поведение как особенность — так как аргументы с изменяемыми стандартными значениями предохраняют свое состояние между вызовами функции, они могут исполнять часть тех же ролей, что и статические локальные переменные функций в языке С. В известной мере они работают во многом подобно глобальным переменным, но их имена локальны по отношению к функциям и потому не конфликтуют с именами в других местах программы.

Тем не менее, для других это выглядит как затруднение, особенно когда они впервые с ним сталкиваются. В Python существуют более эффективные способы предохранения состояния между вызовами (например, применение замыканий вложенных областей видимости, с которыми мы встречались ранее в данной части, и классов, которые будут исследоваться в части VI).

Кроме того, изменяемые стандартные значения нелегко запомнить (и вообще понять). Они зависят от распределения во времени моментов создания объектов для стандартных значений. В предыдущем примере был только один объект списка для стандартного значения — он создавался при выполнении `def`. Каждый раз, когда функция вызывается, вы не получаете новый список, поэтому существующий список растет с каждым новым добавлением к нему элемента; он не сбрасывается в пустое состояние при каждом вызове.

Если вы заинтересованы в другом поведении, тогда просто создайте копию стандартного значения в начале тела функции или перенесите выражение для стандартного значения внутрь тела функции. Так как значение находится в коде, который действительно выполняется при каждом запуске функции, вы будете получать каждый раз новый объект:

```
>>> def saver(x=None):
    if x is None:      # Аргумент не передавался?
        x = []         # Каждый раз выполнять код для создания нового объекта списка
        x.append(1)     # Изменяет новый объект списка
    print(x)

>>> saver([2])
[2, 1]
>>> saver()           # Здесь список не растет
[1]
>>> saver()
[1]
```

Кстати, оператор `if` в рассматриваемом примере можно было бы *почти* заменить присваиванием `x = x or []`, которое извлекает преимущество из того факта, что операция `or` в Python возвращает один из объектов своих operandов: если аргумент не передавался, то `x` получает стандартное значение `None`, поэтому `or` возвратит новый пустой список в правой части операции.

Однако мы получаем не совсем то же самое. В случае передачи пустого списка выражение `or` приведет к тому, что функция расширит и возвратит заново созданный список, а не расширит и возвратит переданный список подобно версии с оператором `if`. (Выражение приобретает вид `[] or []` и результатом его вычисления будет новый пустой список в правой части; см. раздел “Значения истинности и булевские проверки” в главе 12, если не можете вспомнить почему.) Требования в реальных программах могут диктовать любое из двух линий поведения.

В наши дни еще один способ достижения эффекта с предохранением изменяемых объектов, выбранных для стандартных значений, который вероятно меньше сбивает с толку, предусматривает использование *атрибутов функций*, обсуждавшихся в главе 19:

```
>>> def saver():
    saver.x.append(1)
    print(saver.x)

>>> saver.x = []
>>> saver()
[1]
>>> saver()
[1, 1]
>>> saver()
[1, 1, 1]
```

Имя функции является глобальным по отношению к самой функции, но объявлять его не нужно, т.к. оно напрямую не изменяется внутри функции. При подобной записи присоединение объекта к функции оказывается гораздо более явным (и возможно не настолько магическим).

Функции без операторов `return`

Операторы `return` (и `yield`) в функциях Python необязательны. Когда функция не возвращает значение явно, то она завершает работу, когда управление выходит за конец тела функции. Формально все функции возвращают какое-то значение; если не снабдить функцию оператором `return`, тогда она автоматически будет возвращать объект `None`:

```
>>> def proc(x):
    print(x)      # Отсутствие return означает возвращение None
>>> x = proc('testing 123...')
testing 123...
>>> print(x)
None
```

Функции подобного рода, не содержащие `return`, представляют собой эквивалент Python того, что в ряде языков называется “процедурами”. Они обычно вызываются как операторы, а результаты `None` игнорируются, т.к. работа делается без вычисления полезного результата.

Об этом стоит знать, потому что Python не будет сообщать о том, что вы пытаетесь потребить результат функции, в которой отсутствуют операторы `return`. Скажем, как отмечалось в главе 11, присваивание результата спискового метода `append` не приводит к возникновению ошибки, но вы получите обратно `None`, а не модифицированный список:

```
>>> list = [1, 2, 3]
>>> list = list.append(4)      # append - это "процедура"
>>> print(list)              # append изменяет список на месте
None
```

В разделе “Распространенные затруднения при написании кода” главы 15 данный аспект обсуждался более подробно. В общем случае любые функции, которые выполняют свою работу в форме побочного эффекта, обычно спроектированы для запуска как операторы, а не выражения.

Прочие затруднения, связанные с функциями

Существуют два дополнительных связанных с функциями затруднения, обзор которых уже приводился, но они достаточно распространены, чтобы заслуживать повторения.

Объемлющие области видимости и переменные цикла: фабричные функции

Мы упоминали об этом затруднении в главе 17 при обсуждении областей видимости объемлющих функций, но напомним: при написании фабричных функций (также известных как замыкания) не полагайтесь на поиск в областях видимости объемлющих функций в отношении переменных, которые изменяются посредством циклов. Когда сгенерированная функция позже вызывается, все ссылки такого рода будут помнить значение на *последней* итерации цикла в области видимости объемлющей функции. В данном случае должны применяться стандартные значения, чтобы сохранять значения переменных цикла, а не полагаться на автоматический поиск в объемлющих

областях видимости. Дополнительные сведения по этой теме представлены в разделе “Переменные цикла могут требовать стандартные значения, а не области видимости” главы 17.

Сокрытие встроенных функций за счет присваивания: экранирование

В главе 17 также была описана возможность повторного присваивания встроенных имен в ближайшей локальной или глобальной области видимости; повторное присваивание фактически скрывает и заменяет такое встроенное имя для оставшейся части кода в области видимости, где присваивание произошло. Это означает, что использовать первоначальное значение для имени не удастся. До тех пор, пока встроенное значение для имени не требуется, проблема не возникает — многие имена являются встроенными и могут свободно применяться многократно. Тем не менее, если выполнить повторное присваивание встроенному имени, на которое полагается ваш код, тогда могут появиться проблемы. Таким образом, либо не поступайте так, либо используйте инструмент вроде *PyChecker*, который способен предупредить о такой ситуации. Хорошая новость в том, что часто применяемые встроенные функции скоро станут второй натурой, а средство улавливания ошибок Python будет предупреждать на ранней стадии тестирования, если ваше встроенное имя оказывается не тем, о чем вы думали.

Резюме

Глава завершила обсуждение функций и встроенных итерационных инструментов проработкой более крупного учебного примера, в котором измерялась производительность альтернативных вариантов итерации и версий Python, а также обзором распространенных заблуждений, связанных с функциями, чтобы помочь избежать ловушек. У истории с итерацией есть последнее продолжение в части VI, при обсуждении перегрузки операций в главе 30 второго тома, где будет показано, как создавать определяемые пользователем итерируемые объекты, которые генерируют значения, с помощью классов и `__iter__`.

Итак, часть книги, посвященная функциям, завершена. В следующей части мы расширим знания о *модулях* — файлах инструментов, которые образуют самую верхнюю организационную единицу в Python и структуру, где всегда находятся наши функции. Затем мы исследуем классы — инструменты, которые по большому счету представляют собой пакеты функций с особыми первыми аргументами. Как выясняется, определяемые пользователем классы могут реализовывать объекты, подключаемые к протоколу итерации в точности как встречавшиеся ранее генераторы и итерируемые объекты. На самом деле все, что мы изучили в этой части книги, будет применяться позже, когда функции всплывают в контексте методов классов.

Но прежде чем переходить к модулям, закрепите пройденный материал, ответив на контрольные вопросы главы, и выполните упражнения для данной части книги, чтобы попрактиковаться с функциями.

Проверьте свои знания: контрольные вопросы

1. Какие выводы вы можете сделать из этой главы об относительной скорости итерационных инструментов Python?
2. Какие выводы вы можете сделать из этой главы об относительной скорости хронометрируемых версий Python?

Проверьте свои знания: ответы

1. В общем случае списковые включения быстрее всех; `map` побеждает списковые включения в Python, только когда все инструменты обязаны вызывать функции, циклы `for` имеют тенденцию быть медленнее включений, а генераторные функции и выражения медленнее включений на постоянную величину. В реализации PyPy некоторые сведения отличаются; например, `map` часто показывает разную относительную производительность, а списковые включения кажутся всегда самыми быстрыми, возможно из-за оптимизаций на уровне функций.

По крайней мере, так обстоят дела на сегодняшний день для протестированных версий Python, на заданном компьютере и с имеющейся разновидностью хронометрируемого кода — результаты могут варьироваться, если какой-то из трех аспектов окажется другим. Для тестирования своих сценариев использования с целью получения более подходящих результатов применяйте любительский модуль `timer` или стандартный библиотечный модуль `timeit`. Также имейте в виду, что итерация — это всего лишь одна составляющая времени программы: чем больше кода, тем более полной будет картина.

2. В общем PyPy 1.9 (реализация Python 2.7) обычно быстрее CPython 2.7, а CPython 2.7 часто быстрее CPython 3.7. В большинстве случаев измерения времени PyPy примерно в 10 раз быстрее CPython, а CPython 2.7 часто быстрее CPython 3.7 на небольшую постоянную величину. В случаях использования численной математики CPython 2.7 может быть в 10 раз быстрее CPython 3.7, а PyPy — в 100 раз быстрее CPython 3.7. В остальных случаях (например, строковые операции и файловые итераторы) реализация PyPy может оказаться в 10 раз медленнее CPython, хотя на некоторые результаты могут повлиять отличия в `timeit` и управлении памятью. Эталонный тест `pystone` подтверждает такое относительное ранжирование по скорости, хотя сообщаемый им размер отличий будет другим из-за хронометрируемого кода.

По крайней мере, так обстоят дела на сегодняшний день для протестированных версий Python, на заданном компьютере и с имеющейся разновидностью хронометрируемого кода — результаты могут варьироваться, если какой-то из трех аспектов окажется другим. Для тестирования своих сценариев использования с целью получения более подходящих результатов применяйте любительский модуль `timer` или стандартный библиотечный модуль `timeit`. Это особенно справедливо, когда измеряется время реализаций Python, которые могут оптимизироваться в каждом новом выпуске.

Проверьте свои знания: упражнения для части IV

В предложенных далее упражнениях вам предстоит приступить к написанию более сложно устроенных программ. Обязательно ознакомьтесь с решениями в приложении и начинайте писать код в файлах модулей. Вряд ли вы захотите набирать код решения упражнений заново в случае допущения ошибки.

1. *Основы.* В интерактивной подсказке Python напишите функцию, которая выводит свой единственный аргумент, и вызовите ее, передавая объекты разнообразных типов: строку, целое число, список, словарь. Затем попробуйте вызвать ее без передачи какого-либо аргумента. Что произошло? Что происходит, когда передаются два аргумента?

2. *Аргументы.* Напишите функцию по имени `adder` в файле модуля Python. Функция должна принимать два аргумента и возвращать сумму (или результат конкатенации) их двух. Затем добавьте в конец файла код для вызова функции `adder` с объектами различных типов (две строки, два списка, два числа с плавающей точкой) и запустите файл как сценарий в окне командной строки системы. Требуется ли явно выводить результаты, чтобы видеть их на экране?
3. *Переменное количество аргументов.* Обобщите написанную в предыдущем упражнении функцию `adder`, чтобы она вычисляла сумму произвольного количества аргументов, и измените вызовы для передачи более или менее двух аргументов. Какой тип имеет возвращаемое значение суммы? (Подсказки: срез `S[:0]` возвращает пустую последовательность того же типа, что и `S`, а с помощью встроенной функции `type` можно проверять типы, но более простой подход ищите во вручную написанных примерах функции `min` в главе 18.) Что происходит при передаче аргументов отличающихся типов? Как насчет передачи словарей?
4. *Ключевые аргументы.* Измените функцию `adder` из упражнения 2, чтобы она принимала и выполняла суммирование/конкатенацию трех аргументов: `def adder(good, bad, ugly)`. Снабдите каждый аргумент стандартным значением и поэкспериментируйте с вызовом функции в интерактивной подсказке. Попробуйте передать один, два, три и четыре аргумента. Работает ли вызов `adder(ugly=1, good=2)`? Почему? Затем обобщите новую функцию `adder` с целью приема и выполнения суммирования/конкатенации для *произвольного* количества ключевых аргументов. Похожая работа делалась в упражнении 3, но здесь необходимо проходить по словарю, а не по кортежу. (Подсказка: метод `dict.keys` возвращает список, по которому можно проходить с помощью цикла `for` или `while`, но для индексации в Python 3.X не забудьте поместить его внутрь вызова `list`; также может помочь метод `dict.values`.)
5. *Словарные инструменты.* Напишите функцию по имени `copyDict(dict)`, которая копирует свой аргумент типа словаря. Она должна возвращать новый словарь, содержащий все элементы из своего аргумента. Используйте для итерации словарный метод `keys` (или в Python 2.2 и последующих версиях проходите по словарю, не вызывая `keys`). Копировать последовательности легко (`X[:]` создает копию верхнего уровня); работает ли такой прием также для словарей? Как объясняется в решении этого упражнения, поскольку словари теперь снабжены похожими инструментами, текущее и следующее упражнения являются лишь тренировкой в написании кода, но все-таки служат характерными примерами функций.
6. *Словарные инструменты.* Напишите функцию по имени `addDict(dict1, dict2)`, которая вычисляет объединение двух словарей. Она должна возвращать новый словарь, содержащий все элементы из своих двух аргументов (по предположению являющиеся словарями). Если в обоих аргументах встречается один и тот же ключ, выбирайте значение из любого по своему усмотрению. Протестируйте написанную функцию, запустив файл как сценарий. Что произойдет, если передать списки вместо словарей? Как можно было бы обобщить функцию для обработки такого случая? (Подсказка: примените упомянутую ранее встроенную функцию `type`.) Имеет ли значение порядок, в котором передаются аргументы?
7. *Дополнительные примеры сопоставления аргументов.* Определите следующие шесть функций (в интерактивной подсказке или в модуле, который можно импортировать):

```

def f1(a, b): print(a, b)                      # Нормальные аргументы
def f2(a, *b): print(a, b)          # Переменное количество позиционных аргументов
def f3(a, **b): print(a, b)      # Переменное количество ключевых аргументов
def f4(a, *b, **c): print(a, b, c)    # Смешанные режимы
def f5(a, b=2, c=3): print(a, b, c)  # Стандартные значения
def f6(a, b=2, *c): print(a, b, c)  # Стандартные значения и переменное
                                    # количество позиционных аргументов

```

Протестируйте приведенные далее вызовы в интерактивной подсказке и попробуйте объяснить каждый результат; в некоторых случаях, возможно, придется обратиться за помощью к алгоритму сопоставления, описанному в главе 18. Считаете ли вы смешанные режимы сопоставления в целом хорошей идеей? Можете ли вы придумать ситуации, когда они были бы полезными?

```

>>> f1(1, 2)
>>> f1(b=2, a=1)
>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)
>>> f5(1)
>>> f5(1, 4)
>>> f6(1)
>>> f6(1, 3, 4)

```

8. *Снова простые числа.* Вспомните показанный ниже фрагмент кода из главы 13, который упрощенно определял, является ли положительное целое число простым:

```

x = y // 2                                # Для значений y > 1
while x > 1:
    if y % x == 0:                         # Остаток от деления
        print(y, 'has factor', x)           # Имеет сомножитель
        break                               # Пропуск else
    x -= 1
else:                                         # Нормальный выход
    print(y, 'is prime')                  # Является простым

```

Упакуйте данный код как многократно используемую функцию в файле модуля (имя у должно быть передаваемым аргументом) и добавьте в конец файла несколько вызовов функции. Пока вы делаете это, поэкспериментируйте с заменой операции // в первой строке операцией /, чтобы посмотреть, как настящее деление изменяет поведение операции / в Python 3.X и нарушает работу кода (обратитесь в главу 5, если требуется напоминание). Что можно сделать с отрицательными числами и значениями 0 и 1? Как насчет ускорения работы функции? Вывод должен выглядеть примерно так:

```

13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0

```

9. *Итерации и включения.* Напишите код для построения нового списка, содержащего квадратные корни всех чисел из следующего списка: [2, 4, 9, 16, 25]. Сначала запишите его как цикл `for`, затем как вызов `map`, далее как списковое включение и наконец как генераторное выражение. Для вычисления квадратного корня применяйте функцию `sqrt` из встроенного модуля `math` (например, импортируйте `math` и вызовите `math.sqrt(x)`). Какой из четырех подходов вам представляется наилучшим?
10. *Измерение времени выполнения инструментов.* В главе 5 были показаны три способа вычисления квадратного корня: `math.sqrt(X)`, `X ** .5` и `pow(X, .5)`. Если в ваших программах присутствует много таких вычислений, тогда может стать важной их относительная производительность. Чтобы выяснить, какой способ является самым быстрым, переделайте сценарий `timerseqs.py`, написанный ранее в главе, чтобы измерить время каждого из этих трех инструментов. При тестировании используйте функции `bestof` или `bestoftotal`, определенные в одном из модулей `timer` данной главы (вы можете применять либо первоначальный вариант с аргументами, передаваемыми только по ключевым словам, который поддерживается лишь в Python 3.X, либо версию для Python 2.X/3.X, а также использовать модуль `timeit` из Python). Вы также можете принять решение перепаковать код тестирования в этом сценарии, чтобы повысить удобство многократного применения — скажем, передавать универсальной тестовой функции кортеж функций, подлежащих тестированию (в текущем упражнении вполне достаточно подхода с копированием и модификацией кода). Какой из имеющихся трех инструментов для вычисления квадратного корня выполняется быстрее всех на вашем компьютере и версии Python? Как бы вы измеряли скорость выполнения включений словарей по сравнению с циклами `for` в интерактивной подсказке?
11. *Рекурсивные функции.* Напишите простую рекурсивную функцию по имени `countdown`, которая выводит числа в ходе обратного отсчета до нуля. Например, вызов `countdown(5)` должен вывести 5 4 3 2 1 stop. Нет очевидных причин решать задачу с помощью явного стека или очереди, но как насчет подхода без использования функции? Имеет ли смысл здесь применять генератор?
12. *Вычисление факториалов.* И напоследок классика компьютерных наук (но оттого не менее иллюстративная). При обсуждении перестановок в главе 20 мы использовали понятие факториалов: $N!$, вычисляемое как $N * (N-1) * (N-2) * \dots * 1$. Например, $6!$ равно $6 * 5 * 4 * 3 * 2 * 1$, или 720. Напишите и хронометрируйте четыре функции, каждая из которых для вызова `fact(N)` возвращает $N!$. Реализуйте эти четыре функции (1) как рекурсивный обратный отсчет согласно главе 19; (2) применяя вызов функции `reduce` согласно главе 19; (3) с помощью простого цикла с подсчетом согласно главе 13; (4) используя библиотечный инструмент `math.factorial` согласно главе 20. Для измерения времени выполнения каждой функции применяйте модуль `timeit` из главы 21. Какие выводы вы смогли сделать из полученных результатов?

ЧАСТЬ V

Модули и пакеты

Модули: общая картина

В этой главе мы начинаем детальное обсуждение *модулей* Python – единиц организации программ наивысшего уровня, которые упаковывают программный код и данные для многократного использования и предоставляют изолированные пространства имен, сводящие к минимуму конфликты имен переменных внутри программ. Выражаясь более конкретно, модули обычно соответствуют файлам программ Python. Каждый файл является модулем, а модули импортируют другие модули, чтобы задействовать определяемые в них имена. Модули могут также соответствовать расширениям, написанным на внешнем языке, таком как C, Java или C#, и даже каталогам в импортированных пакетах. Модули обрабатываются с помощью двух операторов и одной важной функции.

<code>import</code>	Позволяет клиенту (импортеру) извлечь модуль как единое целое.
<code>from</code>	Позволяет клиентам извлекать отдельные имена из модуля.
<code>imp.reload</code> (<code>reload</code> в Python 2.X)	Предоставляет способ перезагрузки кода модуля, не останавливая Python.

Основы модулей были изложены в главе 3, и с тех пор мы часто применяли их. Цель здесь в том, чтобы расширить уже известные базовые концепции модулей и заняться исследованием более развитых способов использования модулей. В данной главе мы повторим основы модулей и выясним роль модулей в общей структуре программ. В последующих главах мы займемся написанием кода согласно изложенной теории.

Попутно мы восполним недостающие детали о модулях – вы узнаете о перезагрузке, атрибутах `__name__` и `__all__`, импортировании пакетов, синтаксисе относительного импортирования, пакетах пространств имен, появившихся в версии Python 3.3, и т.д. Поскольку модули и классы на самом деле являются знаменитыми *пространствами имен*, мы также формально представим соответствующие концепции.

Для чего используются модули?

Выражаясь кратко, модули предоставляют простой способ организации компонентов в систему, выступая в качестве изолированных пакетов переменных, которые известны как *пространства имен*. Все имена, определенные на верхнем уровне файла мо-

дуля, становятся атрибутами объекта импортированного модуля. Как было показано ранее, импортование обеспечивает доступ к именам в глобальной области видимости модуля. Таким образом, глобальная область видимости файла модуля *превращается* в пространство имен объекта модуля, когда файл модуля импортируется. В конечном итоге модули Python позволяют связывать индивидуальные файлы в более крупную программную систему.

В частности, модули исполняют, по крайней мере, три роли.

Многократное использование кода

Как обсуждалось в главе 3, модули дают возможность сохранять код в файлах на постоянной основе. В отличие от кода, набираемого в интерактивной подсказке Python, который исчезает после выхода из Python, код в файлах модулей *постоянен* – его можно перезагружать и повторно запускать столько раз, сколько нужно. Не менее важно и то, что модули представляют собой место для определения имен, известных как *атрибуты*, на которые могут ссылаться многочисленные внешние клиенты. При правильном применении в результате поддерживается *модульная конструкция* программ, группирующая функциональность в многократно используемые единицы.

Разбиение пространства имен системы

Модули также считаются организационной единицей наивысшего уровня в программах Python. Хотя по существу модули – всего лишь пакеты имен, они являются *изолированными* – вы не сможете увидеть имя из другого файла, пока явно не импортируете этот файл. Во многом подобно локальным областям видимости функций такое решение помогает избежать конфликтов имен в программах. На самом деле вам даже не удастся обойти данную особенность – абсолютно все “существует” в модуле, и запускаемый код, и создаваемые объекты всегда неявно заключаются в модули. По указанной причине модули представляют собой естественные инструменты для группирования компонентов системы.

Реализация разделяемых служб или данных

С эксплуатационной точки зрения модули также удобны для реализации компонентов, которые разделяются в рамках системы и потому требуют только *одной копии*. Например, если необходимо предоставить глобальный объект, применяемый в нескольких функциях или файлах, тогда его можно реализовать в модуле, который затем будет импортироваться многими клиентами.

Чтобы по-настоящему понять роль модулей в системе Python, понадобится отвлечься на какое-то время и выяснить общую структуру программы Python.

Архитектура программы Python

До сих пор в книге при описании программ Python я преуменьшал некоторые сложности. На практике программы обычно включают в себя более одного файла. Для всех сценариев кроме самых простых программы будут принимать форму *многофайловых* систем, что демонстрировали программы для измерения времени выполнения кода в предыдущей главе. Даже если вы сумеете уместить свой код в единственный файл, то все равно почти наверняка прибегнете к использованию внешних файлов, которые уже написали другие.

В текущем разделе будет представлена общая *архитектура* программы Python – способ разделения программы на коллекцию файлов исходного кода (также известных как модули) и связывания частей в единое целое. Вы увидите, что Python стимулирует модульную структуру программ, которая группирует функциональность в связанные и многократно применяемые единицы естественными и почти автоматическими способами. Одновременно мы также исследуем основные концепции модулей Python, импортирования и атрибутов объектов.

Структурирование программы

На самом базовом уровне программа Python состоит из текстовых файлов, содержащих *операторы* Python, с одним главным файлом *верхнего уровня* и нулем или большим количеством добавочных файлов, известных как *модули*.

Посмотрим, как все работает. Файл верхнего уровня (он же сценарий) содержит главный поток управления программы – именно данный файл запускается для старта приложения. Файлы модулей являются библиотеками инструментов, используемых для сбора компонентов, которые применяются файлом верхнего уровня и возможно где-нибудь еще. Файлы верхнего уровня используют инструменты, определенные в файлах модулей, а модули применяют инструменты, определенные в других модулях.

Несмотря на то что файлы модулей тоже относятся к файлам кода, они обычно ничего не делают, когда запускаются напрямую; взамен в них определяются инструменты, предназначенные для использования в других файлах. Файл *импортирует* модуль для получения доступа к определенным в нем инструментам, которые известны как его *атрибуты* – имена переменных, присоединяемые к таким объектам, как функции. В конечном счете, для работы с инструментами мы импортируем модули и обращаемся к их атрибутам.

Импортирование и атрибуты

Давайте перейдем к чуть большей конкретике. На рис. 22.1 показана упрощенная структура программы Python, состоящей из трех файлов: *a.py*, *b.py* и *c.py*. Файл *a.py* выбран в качестве файла верхнего уровня; он будет простым текстовым файлом с операторами, которые при его запуске выполняются от начала до конца. Файлы *b.py* и *c.py* – это модули; они являются простыми текстовыми файлами, также содержащими операторы, но их операторы, как правило, не выполняются напрямую. Как объяснялось ранее, модули обычно импортируются другими файлами, которые заинтересованы в применении инструментов, определенных в модулях.

Предположим, что в файле *b.py* на рис. 22.1 определена функция по имени *spam*, предназначенная для внешнего применения. Согласно тому, что уже известно в результате исследования функций в части IV, файл *b.py* будет содержать оператор *def* языка Python для создания объекта функции, который позже можно запускать, передавая ему ноль или большее количество значений в круглых скобках после имени функции:

```
def spam(text):          # Файл b.py
    print(text, 'spam')
```

Теперь допустим, что в *a.py* желательно использовать *spam*. Для этой цели файл может содержать операторы Python следующего вида:

```
import b                # Файл a.py
b.spam('gumby')         # Выводит gumby spam
```

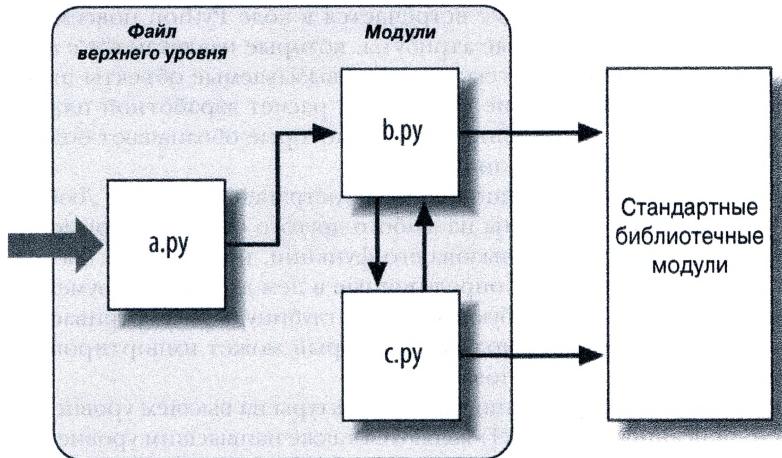


Рис. 22.1. Архитектура программы Python. Программа – это система модулей. Она имеет один файл сценария верхнего уровня (запускаемый для старта программы) и множество файлов модулей (импортируемых библиотек инструментов). Сценарии и модули являются текстовыми файлами, содержащими операторы Python, хотя операторы в модулях обычно лишь создают объекты для использования в более позднее время. Стандартная библиотека Python предоставляет коллекцию предварительно написанных модулей

Первый из них, оператор `import`, предоставляет файлу `a.py` доступ ко всему тому, что определено кодом верхнего уровня в файле `b.py`. Вот что примерно означает код `import b`:

Загрузить файл `b.py` (если он еще не был загружен) и открыть доступ ко всем его атрибутам через имя `b`.

Для удовлетворения указанным целям операторы `import` (и как будет показано позже операторы `from`) выполняют и загружают другие файлы по запросу. Выражаясь более формально, в Python межфайловое связывание модулей не распознается до тех пор, пока не запустятся операторы `import` во время выполнения; их совокупным эффектом будет присваивание именам модулей – простым переменным вроде `b` – объектов загруженных модулей. Фактически имя модуля, применяемое в операторе `import`, служит двум целям: оно идентифицирует внешний файл, подлежащий загрузке, и также становится *переменной*, которой присваивается объект загруженного модуля.

Аналогичным образом объекты, *определяемые* модулем, также создаются во время выполнения как результат запуска `import`: оператор `import` буквально выполняет операторы в целевом файле по одному за раз, чтобы создать их содержимое. В ходе дела каждое имя, присвоенное на верхнем уровне файла, становится атрибутом модуля, доступным импортерам. Скажем, второй оператор в `a.py` вызывает функцию `spam`, определенную в модуле `b` (созданную за счет выполнения оператора `def` во время импортирования), с использованием формы записи для атрибутов объектов. Вот что означает код `b.spam`:

Извлечь значение имени `spam`, которое находится внутри объекта `b`.

В рассматриваемом примере `spam` является вызываемой функцией, так что мы передаем строку в круглых скобках (`'gumby'`). Если вы наберете код приведенных файлов, сохраните их и запустите `a.py`, то будет выведена строка `gumby spam`.

Форма записи `объект.атрибут` встречается в коде Python повсеместно – большинство объектов имеют полезные атрибуты, которые извлекаются с помощью операции точки (.). Одни атрибуты ссылаются на вызываемые объекты наподобие функций, предпринимающих действие (например, расчет заработной платы), а другие представляют собой просто значения данных, которые обозначают более статичные объекты и свойства (скажем, имя лица).

Идея импортирования также широко распространена в Python. Любой файл способен импортировать инструменты из любого другого файла. Например, файл `a.py` может импортировать `b.py` для вызова его функции, но `b.py` мог бы также импортировать `c.py`, чтобы применять определенные в нем другие инструменты. Цепочки импортирования могут иметь любую желаемую глубину: в рассматриваемом примере модуль `a` может импортировать модуль `b`, который может импортировать модуль `c`, который может импортировать снова `b`, и т.д.

Помимо обеспечения организационной структуры на высшем уровне модули (и пакеты модулей, описанные в главе 24) являются также наивысшим уровнем *многократного использования кода* в Python. Помещение кода компонентов в файлы модулей делает их полезными в вашей первоначальной программе и в любых других программах, которые вы можете написать позже. Скажем, если после написания программы на рис. 22.1 обнаружится, что функция `b.spam` оказалась универсальным инструментом, тогда ее можно повторно применить в совершенно другой программе; все, что понадобится сделать – снова импортировать файл `b.py` в файлах другой программы.

Стандартные библиотечные модули

Обратите внимание на прямоугольник в правой части рис. 22.1. Определенные модули, которые ваши программы будут импортировать, предоставляются самим Python, а не берутся из создаваемых вами файлов.

В состав Python входит крупная коллекция служебных модулей, известная как *стандартная библиотека*. Эта коллекция, насчитывающая свыше 300 модулей, содержит независимую от платформы поддержку для распространенных задач программирования: интерфейсы к операционной системе, постоянство объектов, сопоставление текста с образцом, написание сценариев для работы с сетями и Интернетом, построение графических пользовательских интерфейсов и многие другие. Ни один из таких инструментов не является частью самого языка Python, но вы можете их использовать путем импортирования подходящих модулей в любой стандартной копии Python. Поскольку они представляют собой стандартные библиотечные модули, вы также имеете уверенность в том, что они будут доступными и работать переносимым образом на большинстве платформ, где запускается Python.

В примерах книги задействовано несколько стандартных библиотечных модулей (наподобие `timeit`, `sys` и `os` в коде предыдущей главы), но здесь мы лишь поверхностно коснемся тематики, связанной с библиотеками. За исчерпывающим описанием вы должны обратиться в справочное руководство по стандартной библиотеке Python, доступное либо в онлайновом режиме на веб-сайте <http://www.python.org>, либо в рамках установленной копии Python (через IDLE или группу Python в меню Пуск в среде Windows). Обсуждаемый в главе 15 инструмент *PyDoc* предлагает еще один способ исследования стандартных библиотечных модулей.

Из-за настолько большого количества модулей это действительно единственный путь к получению представления о том, какие инструменты имеются в наличии. Вы также найдете руководства по библиотечным инструментам Python в книгах, посвященных программированию на прикладном уровне, таких как *Programming Python*.

(<http://shop.oreilly.com/product/9780596158118.do>) издательства O'Reilly, но онлайновые руководства бесплатны, могут просматриваться в любом веб-браузере (в формате HTML), доступны в других форматах (скажем, в виде справочных файлов Windows) и обновляются каждый раз, когда выходит новый выпуск Python. Дополнительные подсказки ищите в главе 15.

Как работает импортирование

В предыдущем разделе речь шла об импортировании модулей без пояснения, что при этом происходит. Поскольку импортирование лежит в основе структуры программы Python, в данном разделе мы погрузимся в более формальные детали операции импортирования, чтобы сделать процесс менее абстрактным.

Некоторым программистам на языке C нравится сравнивать операции импортирования модулей с директивами `#include` языка C, но на самом деле поступать так не следует; импортирование в Python – не просто вставка одного текста внутрь другого. Они действительно являются операциями времени выполнения, которые делают три отдельных шага, когда программа впервые импортирует заданный файл.

1. *Ищут* файл модуля.
2. *Компилируют* его в байт-код (при необходимости).
3. *Выполняют* код модуля для создания объектов, которые в нем определены.

Для лучшего понимания импортирования модулей мы исследуем перечисленные шаги по очереди. Имейте в виду, что все три шага проделываются только при *первом* импортировании модуля во время выполнения программы; последующее импортирование того же самого модуля при выполнении программы обходит все шаги и просто извлекает из памяти уже загруженный объект модуля. Формально Python хранит загруженные модули в таблице по имени `sys.modules` и просматривает ее в начале операции импортирования. Если модуль в таблице отсутствует, тогда запускается процесс из трех шагов.

1. Поиск файла модуля

Первым делом Python должен найти файл модуля, на который производится ссылка в операторе `import`. Обратите внимание, что в операторе `import` из примера в предыдущем разделе указано имя файла без расширения `.py` и без пути к его каталогу: оператор выглядел как `import b`, а не что-нибудь вроде `import c:\dir1\b.py`. Сведения о пути и расширении опускаются намеренно; взамен для нахождения файла модуля, соответствующего оператору `import`, Python применяет стандартный *путь поиска модулей* и известные типы файлов¹.

¹ Включать путь и расширение в стандартный оператор `import` запрещено правилами синтаксиса. Однако *импортирование пакетов*, которое мы обсудим в главе 24, разрешает операторам `import` содержать часть пути к каталогу, ведущему к файлу, в виде комплекта имен, разделенных точками. Тем не менее, при нахождении самого левого каталога в пути к пакету импортирование пакетов по-прежнему полагается на нормальный путь поиска модулей (т.е. путь к пакету относителен к какому-то каталогу из пути поиска). Кроме того, в операторах `import` не допускается использовать синтаксис указания каталогов, специфичный для платформы; такой синтаксис работает только в пути поиска. Также имейте в виду, что вопросы, связанные с путем поиска файлов, неактуальны в случае запуска *фиксированных двоичных файлов* (обсуждаемых в главе 2), которые обычно встраивают байт-код в двоичный образ.

Поскольку это главная часть операции импортирования, о которой программисты должны знать, вскоре мы возвратимся к данной теме.

2. Компиляция файла модуля (возможная)

После того, как посредством обхода пути поиска модулей найден файл исходного кода, соответствующий оператору `import`, Python при необходимости компилирует его в байт-код. Мы кратко обсуждали байт-код в главе 2, но процесс несколько интереснее, чем объяснялось там. Во время операции импортирования Python проверяет время модификации файлов исходного кода и байт-кода, а также номер версии Python байт-кода, чтобы решить, как поступать. В первом случае используются “отметки времени” файлов, а во втором в зависимости от выпуска Python применяется либо “магическое” число, встроенное в байт-код, либо имя файла. Действие выбирается следующим образом.

Компилировать

Если файл байт-кода *старше* файла исходного кода (т.е. исходный код был модифицирован) или был создан другой *версией* Python, тогда Python автоматически заново генерирует байт-код при запуске программы.

Как будет обсуждаться далее, в Python 3.2 и последующих версиях данная модель несколько изменилась – файлы байт-кода вынесены в отдельный подкаталог `_rucsache_` и содержат в своих именах версию Python во избежание конфликтов и перекомпиляции, когда в системе установлено множество версий Python. В результате устраняется необходимость в проверке номеров версий в байт-коде, но проверка отметок времени по-прежнему используется для обнаружения изменений в исходном коде.

Не компилировать

С другой стороны, если Python обнаруживает файл байт-кода `.rus`, который *не старше* соответствующего файла исходного кода `.ru` и создан той же версией Python, тогда он пропускает шаг компиляции исходного кода в байт-код.

Вдобавок, когда Python находит в пути поиска модулей только файл байт-кода, но не файл исходного кода, он просто напрямую загружает байт-код; таким образом, вы можете поставлять программу в виде только файлов байт-кода, не отправляя файлы исходного кода. Другими словами, шаг компиляции пропускается, если возможно ускорение начального запуска программы.

Обратите внимание, что компиляция происходит во время импортирования файла. Из-за этого вы обычно не будете видеть файл байт-кода `.rus` для файла *верхнего уровня* своей программы, если только он также не импортируется где-то в другом месте – лишь импортированные файлы оставляют после себя файлы `.rus` на компьютере. Байт-код файлов верхнего уровня применяется внутренне и отбрасывается; байт-код импортированных файлов сохраняется в файлах для ускорения будущих операций импортирования.

Файлы верхнего уровня часто спроектированы для выполнения напрямую и не импортируются вообще. Позже мы увидим, что можно спроектировать файл, который будет служить как кодом верхнего уровня программы, так и модулем инструментов, предназначенных для импортирования. Такой файл может выполняться и импортироваться, а потому для него создается файл `.rus`. При обсуждении специального атрибута `__name__` и `__main__` в главе 25 будет показано, как это работает.

3. Выполнение файла модуля

На финальном шаге операции импортирования выполняется байт-код модуля. Все операции в файле выполняются по очереди, от начала до конца, и любые присваивания именам на данном шаге генерируют атрибуты результирующего объекта модуля. Так создаются инструменты, определяемые кодом модуля. Скажем, операторы `def` в файле запускаются на стадии импортирования для создания объектов функций и их присваивания атрибутам внутри объекта модуля. Функции затем вызываются в файлах, импортирующих файл модуля.

Из-за того, что последний шаг импортирования фактически выполняет код файла, если любой код верхнего уровня в файле модуля делает реальную работу, то ее результаты будут видны во время импортирования. Например, операторы `print` верхнего уровня в модуле отображают вывод при импортировании файла. Операторы `def` для функций просто определяют объекты для использования в будущем.

Как видите, операции импортирования включают в себя довольно много работы — они ищут файлы, возможно, запускают компилятор и выполняют код Python. По этой причине любой заданный модуль импортируется по умолчанию только *один раз* на процесс. Будущие операции импортирования пропускают все три шага и повторно используют уже загруженный модуль в памяти. Если файл нужно импортировать снова после того, как он был загружен (скажем, для поддержки динамических настроек со стороны конечного пользователя), тогда придется применить вызов `imp.reload` — инструмент, с которым мы встретимся в следующей главе².

Файлы байт-кода: `__pycache__` в Python 3.2+

Ранее кратко упоминалось, что способ, которым Python сохраняет файлы для запоминания байт-кода, полученного в результате компиляции исходного кода, в Python 3.2 и последующих версиях изменился. Прежде всего, если по какой-либо причине Python не удается записать файл, чтобы сохранить байт-код на компьютере, то программа все равно нормально выполнится — Python просто создает и использует байт-код в памяти и по завершении программы отбрасывает его. Однако для ускорения начального запуска он пытается сохранить байт-код в файле, чтобы в следующий раз пропустить шаг компиляции. Способ, которым это делается, зависит от версии Python.

B Python 3.1 и предшествующих версиях (включая все версии Python 2.X)

Байт-код сохраняется в файлах внутри *того же самого каталога*, где находятся соответствующие файлы исходного кода, обычно с расширением `.rus` (например, `module.rus`). Файлы байт-кода также внутренне снабжаются меткой с номером версии Python, в которой они создавались (известной разработчикам как “магическое” поле), поэтому Python известно о том, что они должны быть заново скомпилированы, когда программа запускается под управлением другой версии Python. Скажем, если вы провели модернизацию до новой версии Python, где байт-код отличается, тогда все ваши файлы байт-кода автоматически перекомпилируются из-за несовпадения номеров версий даже при отсутствии каких-либо изменений в исходном коде.

² Как было описано ранее, Python хранит уже импортированные модули во встроенном словаре `sys.modules`, поэтому он в состоянии отслеживать, что было загружено. На самом деле, если вы хотите выяснить, какие модули загружены, то можете импортировать `sys` и вывести `list(sys.modules.keys())`. Другие сценарии применения внутреннего словаря `sys.modules` приводятся в главе 25.

В Python 3.2 и последующих версиях

Байт-код сохраняется в файлах внутри подкаталога по имени `__pycache__`, который Python при необходимости создает и который находится в каталоге, содержащем соответствующие файлы исходного кода. Такая модель помогает избежать беспорядка в каталогах исходного кода за счет вынесения файлов байт-кода в собственный подкаталог. Кроме того, хотя файлы байт-кода по-прежнему получают расширение `.рус`, как было раньше, они имеют более описательные имена, которые включают текстовую идентификацию *версии Python*, создавшей их (например, `module.cpython-32.rус`). В итоге не возникают конфликты и *повторные компиляции*: Поскольку каждая установленная версия Python может иметь собственную уникально именованную версию файлов байт-кода в подкаталоге `__pycache__`, запуск под управлением выбранной версии не приводит к переписыванию одного байт-кода другим и не требует новой компиляции. Формально имена файлов байт-кода также включают *название Python*, который создал их, так что СPython, Jython и другие реализации, упомянутые в предисловии и главе 2, способны сосуществовать на одном компьютере, не мешая работать друг другу (раз они поддерживают такую модель).

В *обоих* моделях Python всегда заново создает файл байт-кода, если вы изменили файл исходного кода с момента последней компиляции, но отличия в версиях обрабатываются по-разному — с применением магических чисел и замены в Python 3.2 и предшествующих версиях и с использованием имен файлов, допускающих множество копий, в Python 3.2 и последующих версиях.

Модели файлов байт-кода в действии

Ниже приведен короткий пример двух моделей в действии под управлением Python 2.X и Python 3.7. Ради экономии пространства большая часть текста, отображаемого командой `dir` в Windows, опущена, а код применяемого здесь сценария не показан, т.к. он не имеет отношения к текущему обсуждению (он взят из главы 2 и просто выводит два значения). *До версии Python 3.2* файлы байт-кода после создания операциями импортирования обнаруживались рядом со своими файлами исходного кода:

```
c:\code\py2x> dir
05/31/2019  10:58 AM           39 script0.py

c:\code\py2x> C:\python27\python
>>> import script0
hello world
1267650600228229401496703205376
>>> ^Z

c:\code\py2x> dir
05/31/2019  10:58 AM           39 script0.py
05/31/2019  11:00 AM          154 script0.rус
```

Тем не менее, *в Python 3.2 и последующих версиях* файлы байт-кода хранятся в подкаталоге `__pycache__` и содержат в своих именах детали о версиях и реализациях Python, чтобы избежать беспорядка и конфликтов между линейками Python, установленными на компьютере:

```
c:\code\py2x> cd ..\py3x
c:\code\py3x> dir
05/31/2019  10:58 AM           39 script0.py
```

```
c:\code\py3x> C:\python33\python
>>> import script0
hello world
1267650600228229401496703205376
>>> ^Z

c:\code\py3x> dir
05/31/2019 10:58 AM           39 script0.py
05/31/2019 11:00 AM    <DIR>      __pycache__  

c:\code\py3x> dir __pycache__
05/31/2019 11:00 AM           184 script0.cpython-33.pyc
```

Критически важно то, что в рамках модели Python 3.2 и последующих версий импортирование того же самого файла с помощью другой версии Python создает *другой* файл байт-кода вместо переписывания *одиночного* файла, как делается в модели, принятой до Python 3.2. В более новой модели каждая версия и реализация Python имеет собственные файлы байт-кода, готовые к загрузке при следующем запуске программы (ранние версии Python будут благополучно использовать свою схему на том же самом компьютере):

```
c:\code\py3x> C:\python32\python
>>> import script0
hello world
1267650600228229401496703205376
>>> ^Z

c:\code\py3x> dir __pycache__
05/31/2019 12:28 PM           178 script0.cpython-32.pyc
05/31/2019 11:00 AM           184 script0.cpython-33.pyc
```

Более новая модель файлов байт-кода Python 3.2 вероятно лучше, т.к. она избегает повторной компиляции, когда на компьютере установлено более одной версии Python – частая ситуация в современном смешанном мире Python 2.X/3.X. С другой стороны, не обошлось без потенциальных несовместимостей в программах, которые полагаются на прежнюю структуру файлов и каталогов. Скажем, проблема несовместимости может возникать в определенных инструментальных программах, хотя большинство правильно ведущих себя инструментов должны работать, как и ранее. Возможное влияние описано в разделе документации Python 3.2, посвященном нововведениям.

Также имейте в виду, что данный процесс полностью *автоматический* – он является побочным эффектом выполнения программ – и большинство программистов, скорее всего, даже не будут заботиться или замечать разницу помимо более быстрого начального запуска из-за меньшего количества повторных компиляций.

Путь поиска модулей

Как упоминалось ранее, большинству программистов обычно придется беспокоиться только о первой части процедуры импортирования – поиске необходимого файла. Из-за того, что может возникнуть потребность сообщить Python, где искать файлы, подлежащие импортированию, вы должны знать, каким образом расширять путь поиска.

Во многих случаях вы можете положиться на автоматическую природу пути поиска модулей для импортирования и вообще не нуждаться в конфигурировании этого пути.

Однако если вы хотите иметь возможность импортировать файлы, определяемые пользователем, через границы каталогов, то должны знать, как работает путь поиска, чтобы настраивать его. Грубо говоря, путь поиска модулей Python формируется объединением следующих главных компонентов, из которых одни установлены заранее, а другие можно подстраивать для указания Python, где проводить поиск.

1. Домашний каталог программы.
2. Каталоги PYTHONPATH (если установлены).
3. Каталоги стандартной библиотеки.
4. Содержимое любых файлов .pth (при их наличии).
5. Подкаталог site-packages, где размещаются сторонние расширения.

В конечном итоге объединение перечисленных пяти компонентов образует `sys.path` – изменяемый список строк с именами каталогов, который будет подробно описан далее в разделе. Первый и третий элементы пути поиска определяются автоматически. Тем не менее, поскольку Python выполняет поиск в объединении этих компонентов с первого до последнего, *второй и четвертый элементы можно применять для расширения пути с целью включения собственных каталогов исходного кода*. Ниже объясняется, как Python использует компоненты пути поиска.

Домашний каталог (устанавливается автоматически)

Сначала Python ищет импортируемый файл в домашнем каталоге. Смысл данного элемента зависит от того, каким образом вы запускаете код. Когда вы запускаете программу, данный элемент представляет собой каталог, который содержит файл сценария верхнего уровня программы. Когда вы работаете в *интерактивной подсказке*, данным элементом является каталог, где производится работа (т.е. текущий рабочий каталог).

Из-за того, что поиск в домашнем каталоге всегда происходит первым, если программа расположена целиком в единственном каталоге, тогда все ее операции импортирования будут работать автоматически, не требуя какого-либо конфигурирования путей. С другой стороны, поскольку поиск в этом каталоге производится первым, его результаты также переопределяются модулями с теми же самыми именами в других местах пути. Будьте осторожны, чтобы случайно не скрыть таким способом библиотечные модули, если они нужны в программе, или применяйте инструменты пакетов, с которыми мы встретимся позже, чтобы частично обойти проблему.

Каталоги PYTHONPATH (допускают конфигурирование)

Затем Python выполняет поиск во всех каталогах, перечисленных в переменной среды PYTHONPATH, слева направо (при условии, что вы вообще установили ее: она не устанавливается автоматически). Выражаясь кратко, PYTHONPATH – просто список определяемых пользователем и специфичных к платформе имен каталогов, которые содержат файлы кода Python. Вы можете добавить все каталоги, из которых должна быть возможность импортирования, и Python расширит путь поиска модулей, включив в него все перечисленные в переменной PYTHONPATH каталоги.

Так как Python производит поиск сначала в домашнем каталоге, установка PYTHONPATH важна только при импортировании файлов за границами каталога, т.е. если нужно импортировать файл, который хранится в каталоге, отличающемся от каталога, где находится файл с операцией импортирования. Возможно, вы захотите устанавливать переменную PYTHONPATH, как только приступите к написанию программ значительного размера. Но до тех пор, пока все файлы модулей хранятся в каталоге, где производится работа (т.е. в домашнем каталоге вроде C:\code, используемого для примеров из книги), импортирование будет функционировать вообще без необходимости заботиться об установке PYTHONPATH.

Каталоги стандартной библиотеки (устанавливаются автоматически)

Далее Python автоматически выполняет поиск в каталогах, в которых установлены стандартные библиотечные модули. Поскольку поиск в них происходит всегда, они обычно не требуют добавления в переменную PYTHONPATH или включения в файлы конфигурации путей (обсуждаются следующими).

Каталоги в файлах конфигурации путей .pth (допускают конфигурирование)

Затем менее употребляемая возможность Python позволяет пользователям добавлять каталоги в путь поиска модулей, просто перечисляя их по одному в строке внутри текстового файла с расширением .pth (от path — путь). Такие файлы конфигурации путей являются несколько более развитым средством, связанным с установкой; мы не будем здесь раскрывать их полностью, но отметим, что они предлагают альтернативу настройкам переменной PYTHONPATH.

Вкратце текстовые файлы с именами каталогов, помещенные в подходящий каталог, могут служить примерно той же роли, что и переменная среды PYTHONPATH. Скажем, если вы работаете в среде Windows с установленной версией Python 3.7, тогда для расширения пути поиска модулей можете поместить файл по имени myconfig.pth на верхний уровень каталога с установленной копией Python (C:\Python37) или в подкаталог site-packages стандартной библиотеки (C:\Python33\Lib\site-packages). В Unix-подобных системах этот файл может находиться в /usr/local/lib/python3.7/site-packages или /usr/local/lib/site-python.

Когда такой файл присутствует, Python будет добавлять каталоги, перечисленные в каждой его строке, с первой по последнюю, ближе к концу списка в пути поиска модулей — в настоящее время после PYTHONPATH и каталогов стандартной библиотеки, но перед каталогом site-packages, куда часто устанавливаются сторонние расширения. На самом деле Python соберет имена каталогов из всех файлов .pth, которые он найдет, и отфильтрует любые дублированные и несуществующие каталоги. Поскольку они являются файлами, а не настройками оболочки, файлы конфигурации путей могут применяться ко всем пользователям установленной копии, а не только к одному пользователю или оболочке. Кроме того, для некоторых пользователей и приложений текстовые файлы могут оказаться проще в записи, чем настройки среды.

Данная возможность сложнее, чем здесь описано. За дополнительными деталями обращайтесь в руководство по библиотеке Python, в особенности к документации по стандартному библиотечному модулю site — этот модуль позволяет настраивать местоположения библиотек Python и файлов конфигурации путей, и в его документации описаны ожидаемые местоположения файлов .pth в целом.

Я рекомендую новичкам использовать переменную PYTHONPATH или может быть одиночный файл .pth, и только в случае, если приходится импортировать между каталогами. Файлы конфигурации путей более часто применяются сторонними библиотеками, которые обычно устанавливают файл .pth в подкаталоге site-packages библиотеки Python, который рассматривается следующим.

Каталог Lib\site-packages сторонних расширений (устанавливается автоматически)

Наконец, Python автоматически добавляет в путь поиска модулей подкаталог site-packages своей стандартной библиотеки. По соглашению он представляет собой место, в которое устанавливается большинство сторонних расширений, часто автоматически посредством служебных инструментов distutils, описанных во врезке “Стороннее программное обеспечение: distutils” далее в главе. Из-за того, что их каталоги установки всегда являются частью пути поиска модулей, клиенты могут импортировать модули таких расширений безо всяких настроек путей.

Конфигурирование пути поиска

Совокупный эффект всего этого заключается в том, что компоненты PYTHONPATH и файлов конфигурации путей делают возможной подстройку мест, в которых операции импортирования ищут файлы. Способ установки переменных среды и места, где хранятся файлы конфигурации путей, варьируется от платформы к платформе. Например, в Windows вы можете использовать компонент Система панели управления, чтобы установить переменную PYTHONPATH в список каталогов, разделенных символами точки с запятой:

```
c:\pycode\utilities;d:\pycode\package1
```

Или же вы могли бы взамен создать текстовый файл по имени C:\Python37\pydirs.pth со следующим содержимым:

```
c:\pycode\utilities
d:\pycode\package1
```

На других платформах настройки аналогичны, но детали могут меняться слишком широко, чтобы раскрыть их в данной главе. За указаниями насчет расширения пути поиска модулей с помощью PYTHONPATH или файлов .pth на различных plataформах обращайтесь в приложение А второго тома.

Вариации пути поиска

Описание пути поиска модулей получилось правильным, но обобщенным; точная конфигурация пути поиска склонна меняться среди платформ, выпусков Python и даже реализаций Python. В зависимости от имеющейся платформы к пути поиска модулей могут также автоматически добавляться дополнительные каталоги.

Скажем, некоторые выпуски Python могут добавлять в путь поиска перед каталогами PYTHONPATH элемент для *текущего рабочего каталога* — каталога, из которого запущена программа. Когда вы запускаете из командной строки, текущий рабочий каталог может отличаться от домашнего каталога файла верхнего уровня (т.е. каталога, где находится файл вашей программы), который добавляется всегда. Поскольку текущий рабочий каталог может меняться каждый раз, когда программа запускается, обычно вы не должны полагаться на его значение при импортировании.

Дополнительные сведения о запуске программ из командных строк приведены в главе 3³.

Чтобы выяснить, как ваша версия Python конфигурирует путь поиска модулей на имеющейся платформе, вы всегда можете просмотреть `sys.path` — тема следующего раздела.

Список `sys.path`

Если вы хотите увидеть, что путь поиска модулей действительно сконфигурирован на компьютере, тогда можете проинспектировать путь в том виде, как он известен Python, выведя встроенный список `sys.path` (т.е. атрибут `path` стандартного библиотечного модуля `sys`). Данный список строк с именами каталогов представляет собой действующий путь поиска внутри Python; для операций импортирования Python выполняет поиск в каждом каталоге в этом списке слева направо и применяет первое найденное файловое совпадение.

Вообще говоря, `sys.path` является путем поиска файлов. Python конфигурирует его при начальном запуске программы, автоматически объединяя домашний каталог файла верхнего уровня (или пустую строку, чтобы обозначить текущий рабочий каталог), любые каталоги `PYTHONPATH`, содержимое любых созданных вами файлов `.pth` и все каталоги стандартной библиотеки. Результатом оказывается список строк и именами каталогов, где Python производит поиск при каждом импортировании нового файла.

Python открывает доступ к списку `sys.path` по двум веским причинам. Во-первых, он предоставляет способ верификации сделанных настроек пути поиска — если вы не видите в списке свои настройки, то еще раз проверьте правильность работы. Например, ниже показано, как выглядит путь поиска модулей в среде Windows и версии Python 3.7 с переменной `PYTHONPATH`, установленной в `C:\code`, и файлом конфигурации путей `C:\Python37\mypath.pth`, в котором указан каталог `C:\Users\someone`. Пустая строка в начале списка означает текущий каталог, с которым объединяются две настройки; остальное — это каталоги и файлы стандартной библиотеки, а также подкаталог `site-packages` для сторонних расширений:

```
>>> import sys
>>> sys.path
 ['', 'C:\\Code', 'F:\\Python37\\Lib\\idlelib',
 'F:\\Python37\\python37.zip', 'F:\\Python37\\DLLs',
 'F:\\Python37\\lib', 'F:\\Python37', 'C:\\Users\\someone',
 'F:\\Python37\\lib\\site-packages']
```

Во-вторых, если вы знаете, что делаете, то список `sys.path` обеспечивает для сценариев способ ручной подстройки их путей поиска. Как будет показано в примере позже в текущей части книги, за счет модификации списка `sys.path` можно изменить путь поиска для всех будущих операций импортирования, выполняемых при запуске программы. Однако такие изменения сохраняются только на протяжении выполнения сценария; переменная `PYTHONPATH` и файлы `.pth` предлагают более постоянные способы модификации пути — первый вариант на каждого пользователя, а второй на каждую установленную копию.

³ Так же дождитесь обсуждения нового *синтаксиса относительного импортирования* и правил поиска в Python 3.X, приведенного в главе 24; они модифицируют путь поиска для операторов `from` в файлах внутри пакетов, когда применяются символы точки (например, `from . import string`). По умолчанию в Python 3.X операции импортирования не предпринимают автоматический поиск в собственном каталоге пакета, если только в самом пакете не используется такое относительное импортирование.

С другой стороны, некоторые программы *действительно* нуждаются в изменении `sys.path`. Скажем, сценарии, запускаемые на веб-серверах, часто выполняются от имени пользователя `nobody` (никто), чтобы ограничить доступ к компьютеру. Из-за того, что такие сценарии обычно полагаются на пользователя `nobody` при установке `PYTHONPATH` каким-то отдельно взятым способом, до выполнения любых операторов `import` они нередко устанавливают `sys.path` вручную для включения обязательных каталогов исходного кода. Как правило, метода `sys.path.append` или `sys.path.insert` будет достаточно, хотя он действует в течение только одиночного запуска программы.

Выбор файла модуля

Имейте в виду, что расширения имен файлов (например, `.py`) намеренно не указываются в операторах `import`. Python выбирает первый найденный в пути поиска файл, имя которого соответствует имени, указанному в операторе `import`. На самом деле операции импортирования представляют собой точку интерфейса к множеству внешних компонентов – исходному коду, разновидностям байт-кода, скомпилированным расширениям и т.п. Python автоматически выбирает любой вариант, дающий совпадение с именем модуля.

Источники модулей

В настоящее время оператор `import` вида `import b` способен загрузить или распознать:

- файл исходного кода по имени `b.py`;
- файл байт-кода по имени `b.pyc`;
- файл оптимизированного байт-кода по имени `b.pyc` (менее распространенный формат);
- каталог по имени `b` для операций импортирования пакетов (рассматриваются в главе 24);
- скомпилированный модуль расширения, написанный на C, C++ или другом языке и динамически связываемый при импортировании (например, `b.so` в Linux либо `b.dll` или `b.pyd` в Cygwin и Windows);
- скомпилированный встроенный модуль, написанный на C и статически связанный с Python;
- компонент в файле ZIP, который автоматически извлекается при импортировании;
- образ в памяти для фиксированных исполняемых файлов;
- класс Java в версии Jython языка Python;
- компонент .NET в версии IronPython языка Python.

Операции импортирования расширений C, классов Java в Jython и пакетов распространяют импортирование за пределы простых файлов. Тем не менее, для импортеров различия в типах загружаемых файлов совершенно несущественны как при импортировании, так и при извлечении атрибутов модулей.

Оператор `import b` обеспечивает получение модуля `b`, чем бы он ни был, в соответствии с путем поиска модулей, а `b.attr` извлекает элемент из модуля, будь он переменной Python или связанной функцией C. Некоторые стандартные модули, используемые в книге, на самом деле написаны на языке C, а не Python; поскольку они выглядят в точности как файлы модулей на Python, их клиентам не приходится беспокоиться об этом факте.

Приоритеты при выборе

При наличии файлов `b.py` и `b.so` в разных каталогах Python всегда будет загружать тот из них, который он найдет в первом (самом левом) каталоге внутри пути поиска модулей во время просмотра списка `sys.path`. Но что произойдет, если Python обнаружит файлы `b.py` и `b.so` в *том же самом* каталоге? В таком случае Python следует стандартному порядку выбора, хотя он и не гарантированно останется одинаковым с течением времени либо между реализациями. В целом вы не должны полагаться на то, какой тип файла Python выберет внутри заданного каталога – назначайте модулям несовпадающие имена или конфигурируйте путь поиска модулей так, чтобы сделать предпочтения выбора модулей явными.

Привязки импортирования и файлы ZIP

Обычно операции импортирования работают так, как описано в этом разделе – они ищут и загружают файлы на компьютере. Однако с применением *привязок импортирования* можно переопределять многое из того, что операции импортирования делают в Python. Привязки можно использовать для того, чтобы заставить операции импортирования предпринимать разную полезную работу, такую как загрузка файлов их архивов, выполнение расшифровки и т.д.

Фактически Python сам задействует такие привязки, чтобы разрешить импортирование файлов прямо из ZIP-архивов: архивированные файлы автоматически извлекаются на стадии импортирования, когда файл `.zip` выбирается из пути поиска импортируемых модулей. Скажем, в наши дни одним из каталогов в списке `sys.path` является файл `.zip`, как было показано выше. За дополнительными деталями обращайтесь к описанию встроенной функции `__import__` в руководстве по стандартной библиотеке Python; функция `__import__` представляет собой настраиваемый инструмент, который в действительности запускают операторы `import`.



Также ознакомьтесь с нововведениями, появившимися в Python 3.3 и последующих версиях, чтобы выяснить, что изменилось в плане импортирования. Вкратце, начиная с версии Python 3.3, функция `__import__` реализована как `importlib.__import__`, отчасти для унификации и чтобы более четко показать ее реализацию.

Второй из двух вызовов также помещен внутрь `importlib.import_module` – инструмента, который согласно текущему руководству по Python в целом предпочтительнее, чем `__import__`, для прямых вызовов импортирования по строке с именем (прием, обсуждаемый в главе 25). В настоящее время работают оба вызова, хотя функция `__import__` поддерживает настраиваемые операции импортирования за счет замены встроенной области видимости (см. главу 17), а другие методики поддерживают подобные роли. Дополнительные подробности ищите в руководстве по стандартной библиотеке Python.

Файлы оптимизированного байт-кода

Наконец, в Python поддерживается понятие файлов оптимизированного байт-кода `.ruo`, которые создаются и запускаются с помощью флага командной строки `-O` интерпретатора Python, а также автоматически генерируются рядом инструментов установки. Но поскольку файлы `.ruo` выполняются лишь слегка быстрее (обычно на 5%) нормальных файлов `.pyc`, они применяются нечасто. Например, система PyPy (см. главы 2 и 21) обеспечивает более существенное ускорение. Дополнительные детали о файлах `.ruo` приведены в приложении А второго тома и в главе 36 (т том 2).

Стороннее программное обеспечение: `distutils`

Приведенное в главе описание настроек пути поиска модулей ориентировано главным образом на исходный код, который вы пишете самостоятельно. Сторонние расширения для Python обычно для своей автоматической установки используют инструменты `distutils` из стандартной библиотеки, так что применение их кода не требует конфигурирования пути.

Системы, которые используют `distutils`, как правило, поступают со сценарием `setup.py`, запускаемым для их установки. Сценарий `setup.py` импортирует и применяет модули `distutils` для размещения таких систем в каталоге, который автоматически является частью пути поиска модулей (обычно в подкаталоге `Lib\site-packages` дерева каталогов установленной копии Python, где бы оно не находилось на целевом компьютере).

Чтобы получить больше сведений о распространении и установке с помощью `distutils`, почитайте стандартный комплект руководств по Python; такие вопросы выходят далеко за рамки этой книги (например, `distutils` вдобавок предлагает способы автоматической компиляции расширений, написанных на языке C, на целевом компьютере). Также обратите внимание на стороннюю систему *Python Eggs* с открытым кодом, которая добавляет проверку зависимостей для установленного программного обеспечения Python.

Примечание. Перед выходом версии Python 3.3 велись разговоры об объявлении пакета `distutils` устаревшим и его замене более новым пакетом `distutils2` в стандартной библиотеке. Тем не менее, несмотря на ожидания, он не появился ни в Python 3.3, ни в текущей на данный момент версии Python 3.7 – похоже, работа над пакетом `distutils2` была приостановлена.

Резюме

В главе были раскрыты основы модулей, атрибутов и импортирования, а также исследовано поведение операторов `import`. Вы узнали, что операция импортирования ищет указанный файл в пути поиска модулей, компилирует его в байт-код и выполняет все имеющиеся внутри операторы для генерации содержимого. Вы также научились конфигурировать путь поиска, чтобы иметь возможность импортировать из каталогов, отличающихся от домашнего каталога и каталогов стандартной библиотеки, главным образом посредством настроек `PYTHONPATH`.

Как демонстрировалось в главе, операция импортирования и модули лежат в самой основе программной архитектуры Python. Более крупные программы разбиваются на множество файлов, которые с помощью импортирования связываются вместе во время выполнения. В свою очередь импортирование использует путь поиска модулей для нахождения файлов, а модули определяют атрибуты для внешнего применения.

Конечно, весь смысл операций импортирования и модулей – снабдить программу структурой, которая разделяет свою логику на изолированные программные компоненты. Код в одном модуле изолирован от кода в другом; на самом деле ни один файл не может даже видеть имена, определенные в другом файле, если только не были выполнены явные операторы `import`. По этой причине модули сводят к минимуму конфликты имен между разными частями программы.

В следующей главе вы увидите, что все описанное означает с точки зрения действительных операторов и кода. Однако прежде чем двигаться дальше, закрепите пройденный материал, ответив на традиционные контрольные вопросы по главе.

Проверьте свои знания: контрольные вопросы

1. Каким образом файл исходного кода модуля становится объектом модуля?
2. Затем вам может понадобиться установка переменной среды PYTHONPATH?
3. Назовите пять главных компонентов в пути поиска импортируемых модулей.
4. Назовите четыре типа файлов, которые Python может загрузить в ответ на операцию импортирования.
5. Что такое пространство имен и что содержит пространство имен модуля?

Проверьте свои знания: ответы

1. Файл исходного кода модуля автоматически становится объектом модуля, когда этот модуль импортируется. Формально на стадии импортирования исходный код модуля выполняется по одному оператору за раз и все имена, присвоенные в процессе, превращаются в атрибуты объекта модуля.
2. Устанавливать переменную среды PYTHONPATH необходимо только при импортировании из каталогов, отличающихся от каталога, в котором вы работаете (т.е. текущего каталога при работе в интерактивной подсказке или каталога, содержащего ваш файл верхнего уровня). На практике это часто встречающийся случай для нетривиальных программ.
3. Пятью главными компонентами в пути поиска импортируемых модулей являются домашний каталог сценария верхнего уровня (каталог, содержащий его), все каталоги, перечисленные в переменной среды PYTHONPATH, каталоги стандартной библиотеки, все каталоги внутри файлов конфигурации путей .pth, находящихся в стандартных местах, и подкаталог site-packages для установленных сторонних расширений. Из всех указанных компонентов программисты могут настраивать переменную PYTHONPATH и файлы .pth.
4. Python может загрузить файл исходного кода (.py), файл байт-кода (.pyc или .pyo), модуль расширения C (например, файл .so в Linux или файл .dll либо .pyd в Windows) или каталог с таким же именем для операций импортирования пакетов. Операции импортирования могут также загружать более экзотические вещи, такие как компоненты файлов ZIP, классы Java под управлением Jython, компоненты .NET под управлением IronPython и статически связанные расширения C, для которых файлы вообще отсутствуют. На самом деле за счет использования привязок импортирования с помощью операций импортирования можно загружать произвольные элементы.
5. Пространство имен представляет собой изолированный пакет переменных, которые известны как *атрибуты* объекта пространства имен. Пространство имен модуля содержит все имена, которым производилось присваивание в коде на верхнем уровне файла модуля (т.е. не внутри операторов def или class). Формально глобальная область видимости модуля *превращается* в пространство имен с атрибутами объекта модуля. Пространство имен модуля также может быть изменено присваиваниями в других файлах, которые его импортируют, хотя обычно это не одобряется (недостатки межфайловых изменений обсуждались в главе 17).

Основы написания модулей

Теперь, когда вам известны общие идеи, лежащие в основе модулей, давайте рассмотрим несколько примеров модулей в действии. Хотя здесь повторяются определенные темы, затронутые в примерах предшествующих глав, вы быстро заметите, что они приводят к дополнительным деталям, касающимся модулей Python, которые пока еще не встречались — вложение, перезагрузка, области видимости и т.д.

Модули Python легко *создавать*; они представляют собой всего лишь файлы программного кода Python, создаваемые с помощью текстового редактора. Не нужно применять какой-то специальный синтаксис, чтобы сообщить Python о том, что создается модуль; им может быть почти любой текстовый файл. Поскольку Python поддерживает все действия, связанные с поиском и загрузкой модулей, модули также легко *использовать*; клиенты просто импортируют модуль или специфические имена, определенные в модуле, и работают с объектами, на которые имена ссылаются.

Создание модулей

Чтобы определить модуль, наберите в своем текстовом редакторе какой-нибудь код Python и сохраните его в текстовом файле с расширением .py; любой файл такого рода автоматически считается модулем Python. Все имена, присвоенные на верхнем уровне модуля, становятся его *атрибутами* (именами, ассоциированными с объектом модуля) и экспортируются для применения клиентами — переменные автоматически превращаются в атрибуты объекта модуля.

Например, если вы поместите следующий оператор def в файл module1.py и импортируете его, то создадите объект модуля с одним атрибутом — именем printer, которое окажется ссылкой на объект функции:

```
def printer(x):      # Атрибут модуля
    print(x)
```

Имена файлов модулей

Прежде чем двигаться дальше, нужно сказать несколько слов об именах файлов модулей. Вы можете называть модули практически как угодно, но имена файлов модулей должны заканчиваться суффиксом .py, если планируется их импортировать. Формально наличие .py необязательно для файлов верхнего уровня, которые будут запускаться, но не импортироваться, однако добавление суффикса .py во всех случаях делает типы файлов более очевидными и дает возможность импортировать любой файл в будущем.

Из-за того, что имена модулей становятся именами переменных внутри программы Python (без суффикса .py), они также обязаны следовать обычным правилам именования переменных, обрисованным в главе 11. Скажем, вы можете создать файл модуля по имени if.py, но будете не в состоянии его импортировать, т.к. if является зарезервированным словом – оператор import if приведет к синтаксической ошибке. На самом деле правилам именования переменных, представленным в главе 11, должны подчиняться как имена *файлов* модулей, так и имена *каталогов*, используемых в операциях импортирования пакетов (обсуждаются в следующей главе); например, они могут содержать только буквы, цифры и подчеркивания. Кроме того, каталоги пакетов также не могут содержать синтаксис, специфичный для платформы, такой как пробелы в своих именах.

Когда модуль импортируется, Python отображает внутреннее имя модуля на внешнее имя файла, добавляя путь к каталогу из пути поиска модулей в начало и .py или другое расширение в конец. Скажем, модуль по имени M в итоге отображается на внешний файл <каталог>\M.<расширение>, который содержит код модуля.

Другие виды модулей

Как упоминалось в предыдущей главе, модуль Python возможно также создавать за счет написания кода на внешнем языке вроде C, C++ и других (например, на Java в реализации Python). Такие модули называются *модулями расшифрований* и, как правило, применяются для помещения в оболочки внешних библиотек, подлежащих использованию в сценариях Python. Когда модули расширений импортируются в коде Python, они выглядят и ведут себя точно так же, как модули, реализованные в виде файлов исходного кода Python – доступ к ним производится посредством операторов import, а сами они предоставляют функции и объекты в форме атрибутов модулей. Модули расширений в этой книге не рассматриваются, так что дополнительные сведения ищите в стандартных руководствах по Python или в книгах, подобных *Programming Python* (<http://www.oreilly.com/catalog/9780596158101>).

Использование модулей

Клиенты могут задействовать только что написанный простой файл модуля за счет выполнения оператора import или from. Оба оператора ищут, компилируют и запускают код файла модуля, если он еще не был загружен. Главное отличие между ними в том, что import извлекает модуль как единое целое, поэтому для извлечения имен модуля они должны уточняться именем самого модуля; в противоположность ему from извлекает (или копирует) специфические имена из модуля.

Давайте посмотрим, как все выглядит в коде. Все последующие примеры приводят к вызову функции printer, определенной в файле модуля module1.py из предыдущего раздела, но различными способами.

Оператор import

В первом примере имя module1 служит двум разным целям – оно идентифицирует внешний файл, подлежащий загрузке, и оно становится переменной в сценарии, которая ссылается на объект модуля после того, как файл загружен:

```
>>> import module1      # Получить модуль как единое целое (один или больше)
>>> module1.printer('Hello world!')    # Уточнить, чтобы получить имена
Hello world!
```

В операторе `import` просто указывается одно или несколько имен модулей для загрузки, разделенные запятыми. Так как оператор `import` дает имя, которое ссылается на *полный объект модуля*, мы обязаны задавать имя модуля, чтобы извлечь его атрибуты (например, `module1.printer`).

Оператор `from`

Напротив, поскольку оператор `from` копирует *специфические имена* из одного файла в другую область видимости, он дает возможность применять скопированные имена в сценарии напрямую, не уточняя их именем модуля (например, `printer`):

```
>>> from module1 import printer      # Копировать переменную (одну или более)
>>> printer('Hello world!')        # Уточнение не требуется
Hello world!
```

Такая форма `from` позволяет указывать одно или несколько имен для копирования, разделенных запятыми. Здесь оператор `from` имеет такой же эффект, как в предыдущем примере, но из-за того, что импортированное имя копируется в область видимости, где находится `from`, использование этого имени в сценарии сопряжено с меньшим объемом набора — мы можем работать с именем напрямую, не задавая включающий модуль. В действительности мы обязаны поступать так; `from` не создает переменную с именем самого модуля.

Как будет поясняться позже, оператор `from` на самом деле является всего лишь незначительным расширением оператора `import` — он импортирует файл модуля обычным образом (в соответствии с трехшаговой процедурой из предыдущей главы), но добавляет дополнительный шаг, который копирует одно или большее количество имен (не объектов) из файла. Загружается целый файл, но вам предоставляются имена для более прямого доступа к его частям.

Оператор `from *`

Наконец, в следующем примере применяется особая форма оператора `from`: когда вместо специфических имен используется `*`, мы получаем копии *всех имен*, присвоенных на верхнем уровне указанного модуля. Затем мы снова можем задействовать в своем сценарии скопированное имя `printer`, не уточняя его именем модуля:

```
>>> from module1 import *      # Копировать все переменные
>>> printer('Hello world!')
Hello world!
```

Формально операторы `import` и `from` вызывают ту же самую операцию импортирования; форма `from *` просто добавляет дополнительный шаг, который копирует все имена из модуля в импортирующую область видимости. Она по существу сворачивает пространство имен одного модуля внутрь другого; опять-таки совокупный эффект для нас — меньший объем набора. Обратите внимание, что `*` работает только в таком контексте; применять сопоставление с образцом для выбора подмножества имен нельзя (хотя с немного большими усилиями можно было бы пройти по атрибуту `__dict__` модуля, который вскоре будет обсуждаться).

Вот и все — использовать модули по-настоящему легко. Тем не менее, чтобы дать вам лучшее представление о том, что действительно происходит, когда модули определяются и применяются, мы зайдемся детальными исследованиями их свойств.



В Python 3.X описанная здесь форма оператора `from ... *` может использоваться *только* на верхнем уровне файла модуля, но не внутри функции. В Python 2.X ее разрешено применять внутри функции, но в любом случае выдается предупреждение. На практике случаи использования оператора `from ... *` внутри функции встречаются редко; его присутствие внутри функции делает невозможным для Python статическое обнаружение переменных до запуска функции. Передовой опыт рекомендует во всех версиях Python перечислять *все* операции импортирования в начале файла модуля; это не требование, но когда они находятся в начале файла, их легче заметить.

Операции импортирования происходят только однократно

Когда новички начинают работать с модулями, то они по обыкновению задают один распространенный вопрос: почему мои операции импортирования перестали работать? Они часто сообщают о том, что первая операция импортирования отработала успешно, но последующие операции импортирования в интерактивном сеансе (или при запуске программы), похоже, не возымели эффекта. На самом деле, какой-либо эффект и не предполагался. В этом разделе объясняется причина.

Модули загружаются и запускаются при выполнении первого оператора `import` или `from` и только первого. Так было задумано — поскольку импортирование является затратной операцией, по умолчанию Python делает его только один раз на файл и однократно на процесс. Более поздние операции импортирования просто извлекают объект уже загруженного модуля.

Код инициализации

В качестве одного последствия, из-за того, что код верхнего уровня в файле модуля обычно выполняется только один раз, вы можете применять его для инициализации переменных. Взгляните на содержимое файла `simple.py`:

```
print('hello')
spam = 1           # Инициализовать переменную
```

В приведенном примере операторы `print` и `=` выполняются, когда модуль импортируется в первый раз, и переменная `spam` инициализируется во время импортирования:

```
% python
>>> import simple      # Первая операция импортирования: загружает
                  # и выполняет код файла
hello
>>> simple.spam      # Присваивание создает атрибут
1
```

Вторая и последующие операции импортирования не выполняют код модуля повторно; они всего лишь извлекают уже созданный объект модуля из внутренней таблицы модулей Python. Таким образом, переменная `spam` не будет инициализироваться заново:

```
>>> simple.spam = 2    # Изменение атрибута в модуле
>>> import simple      # Просто извлекает объект уже загруженного модуля
>>> simple.spam        # Код модуля повторно не выполняется:
                      # атрибут остался неизменившимся
2
```

Разумеется, иногда вы действительно *хотите*, чтобы код модуля выполнился повторно при последующей операции импортирования. Позже в главе мы покажем, как это сделать с помощью функции `reload` в Python.

Операторы `import` и `from` являются присваиваниями

Подобно `def` операторы `import` и `from` являются *исполняемыми*, а не объявлениями на стадии компиляции. Они могут вкладываться внутрь проверок `if` для выбора среди нескольких вариантов; находиться внутри операторов `def` функций, чтобы загружать только по вызову (см. предыдущую врезку “На заметку!”); использоваться в операторах `try` для обеспечения стандартных значений и т.д. Они не распознаются и не запускаются до тех пор, пока Python не встретит их во время выполнения программы. Другими словами, импортируемые модули и имена не будут доступны, пока не выполняются связанные с ними операторы `import` или `from`.

Модификация изменяемых объектов в модулях

Кроме того, как и `def`, операторы `import` и `from` являются *явными присваиваниями*:

- оператор `import` присваивает одиночному имени объект целого модуля;
- оператор `from` присваивает одному или нескольким именам объекты с такими же именами из другого модуля.

Все, что уже обсуждалось относительно присваивания, применимо также к доступу к модулям. Скажем, имена, копируемые с помощью `from`, становятся ссылками на разделяемые объекты. Как и с присваиванием функций, повторное присваивание скопированному имени не оказывает влияния на модуль, из которого было скопировано имя, но модификация разделяемого *изменяемого объекта* через скопированное имя может также изменить его в модуле, откуда оно копировалось. В целях иллюстрации рассмотрим следующий файл `small.py`:

```
x = 1
y = [1, 2]
```

При импортировании посредством `from` мы копируем имена в область видимости импортера, что изначально разделяет объекты, на которые ссылались имена модуля:

```
% python
>>> from small import x, y # Копировать два имени
>>> x = 42                  # Изменяет только локальное имя x
>>> y[0] = 42                # Модифицирует изменяемый объект на месте
```

В данном примере `x` не является разделяемым изменяемым объектом, но `y` является. Имена `y` в импортируемом и импортирующем модулях ссылаются на тот же самый списоковый объект, а потому его модификация в одном месте приводит к изменению в другом:

```
>>> import small           # Получить имя модуля (from этого не позволяет)
>>> small.x                 # x из small - не то же, что x здесь
1
>>> small.y                 # Но мы разделяем модифицированный изменяемый объект
[42, 2]
```

Дополнительные сведения были даны в главе 6. Графическое представление того, что присваивания делаются со ссылками, было предложено на рис. 18.1 (передача аргументов функции); мысленно поменяйте “вызывающий код” и “функция” на “импортируемый модуль” и “импортер”. Результат будет таким же за исключением того, что мы имеем дело с именами в модулях, а не в функциях. Присваивания работают одинаково повсюду в Python.

Межфайловое изменение имен

Вспомните из предыдущего примера, что присваивание `x` в интерактивном сеансе изменяет только имя `x` в этой области видимости, но не `x` в файле – не существует какой-либо связи между именем, скопированным с помощью `from`, и файлом, откуда оно поступило. Чтобы действительно изменить глобальное имя в другом файле, потребуется использовать оператор `import`:

```
% python
>>> from small import x, y      # Скопировать два имени
>>> x = 42                      # Изменяет только локальное имя x
>>> import small                # Получить имя модуля
>>> small.x = 42                 # Изменяет x в другом модуле
```

Явление было впервые представлено в главе 17. Поскольку такого рода изменение переменных в других модулях является распространенным источником путаницы (и часто неудачным проектным решением), мы снова вернемся к данной методике позже в этой части книги. Важно понимать, что изменение `y[0]` в предыдущем сеансе отличается; оно изменяет *объект*, не имя, а имя в обоих модулях ссылается на один и тот же измененный объект.

Эквивалентность `import` и `from`

В предыдущем примере обратите внимание на то, что для доступа к имени модуля `small` вообще мы обязаны выполнить оператор `import` после `from`. Оператор `from` только копирует имена из одного модуля в другой; он не присваивает само имя модуля. По крайней мере, концептуально оператор `from` следующего вида:

```
from module import name1, name2      # Копировать указанные два имени (только)
```

эквивалентен такой последовательности операторов:

```
import module                         # Извлечь объект модуля
name1 = module.name1                   # Копировать имена присваиванием
name2 = module.name2
del module                            # Избавиться от имени модуля
```

Подобно всем присваиваниям оператор `from` создает в импортере новые переменные, которые изначально ссылаются на объекты с теми же самыми именами в импортируемом файле. Однако копируются только имена, но не объекты, на которые они ссылются, и не имя самого модуля. Когда мы применяем форму `from *` данного оператора (`from module import *`), то эквивалентная последовательность остается такой же, но в область видимости импортера копируются все имена верхнего уровня из модуля.

Как видите, на первом шаге оператора `from` запускается нормальная операция `import` со всей семантикой, очерченной в предыдущей главе. По этой причине `from` всегда импортирует *целый* модуль в память, если он еще не был импортирован, независимо от того, сколько имен будет копироваться из файла. Возможность загрузки только части файла модуля (скажем, какой-то одной функции) отсутствует, но из-за того, что модули в Python представлены как байт-код, а не машинный код, то влияние на производительность обычно незначительно.

Потенциальные затруднения, связанные с оператором `from`

Поскольку оператор `from` делает местоположение переменной менее явным и понятным (для читателя кода имя не настолько выразительно, как модуль `.имя`), некоторые пользователи Python рекомендуют большую часть времени использовать `import`, а не `from`. Тем не менее, я не уверен, что такой совет оправдан; оператор `from` обычно широко применяется без чересчур уж многих неприятных последствий. На практике в реалистичных программах часто удобно не набирать имя модуля всякий раз, когда необходимо использовать один из его инструментов. Сказанное особенно справедливо для крупных модулей, которые предоставляют много атрибутов – примером может служить модуль для построения графических пользовательских интерфейсов `tkinter` из стандартной библиотеки.

Правда в том, что оператор `from` обладает потенциалом искажать пространства имен, по крайней мере, в принципе – если вы применяете его для импортирования переменных, у которых оказались такие же имена, как у переменных, существующих в вашей области видимости, тогда эти переменные будут молча переписаны. Такая проблема не возникает с простым оператором `import`, потому что для получения его содержимого вы всегда должны указывать имя модуля (`модуль.атрибут` не будет конфликтовать с переменной по имени `атрибут` в вашей области видимости). Однако до тех пор, пока вы понимаете и ожидаете, что подобное может произойти при использовании `from`, это не является важным вопросом на практике, особенно если вы явно перечисляете импортируемые имена (скажем, `from module import x, y, z`).

С другой стороны, оператор `from` имеет более серьезные проблемы, когда встречается в сочетании с вызовом `reload`, т.к. импортированные имена могут ссылаться на предыдущие версии объектов. Более того, форма `from module import *` на самом деле способна искажать пространства имен и делать имена трудными для понимания, главным образом, когда применяется к более чем одному файлу – в данном случае невозможно выяснить, из какого модуля поступило имя, не считая поиска во внешних файлах исходного кода. В действительности форма `from *` сворачивает одно пространство имен внутри другого и потому сводит на нет саму цель модулей, заключающуюся в разбиении на пространства имен. Мы более детально исследуем указанные проблемы в разделе “Затруднения, связанные с модулями” главы 25.

Вероятно лучший дельный совет здесь – в целом отдавать предпочтение оператору `import` перед `from` для простых модулей, явно перечислять желаемые переменные в большинстве операторов `from` и ограничивать форму `from *` только одной операцией импортирования на файл. В таком случае любые неопределенные имена могут считаться существующими в модуле, на который производится ссылка посредством `from *`. При использовании оператора `from` требуется определенная осторожность, но обладая минимальными знаниями, большинство программистов считут его удобным способом доступа к модулям.

Когда оператор `import` обязателен

Единственный раз, когда вы действительно должны применять `import` взамен `from`, касается случая, где необходимо использовать одно и то же имя, определенное в двух разных модулях. Например, если в двух файлах то же самое имя определено по-разному:

```
# M.py
def func():
    ...делать что-то...
```

```
# N.py
def func():
    ...делать что-то другое...
```

и вам нужно применять в программе обе версии имени, то оператор `from` потерпит неудачу – в области видимости можно иметь только одно присваивание имени:

```
# O.py
from M import func
from N import func    # Перелистывает имя func, извлеченное из M
func()               # Вызывает только N.func!
```

Тем не менее, оператор `import` здесь работает, потому что указание имени включающего модуля делает два имени уникальными:

```
# O.py
import M, N      # Получить модули целиком, не только их имена
M.func()          # Теперь мы можем обращаться к обоим именам
N.func()          # Указание имен модулей обеспечивает уникальность
```

Такой случай довольно необычен, так что вы вряд ли будете его часто встречать на практике. Однако если вы все-таки с ним столкнетесь, тогда оператор `import` позволит избежать конфликта имен. Другой выход из затруднительного положения предусматривает использование расширения `as`, которое мы рассмотрим в главе 25, но оно достаточно простое, чтобы представить его ниже:

```
# O.py
from M import func as mfunc    # Переименование с помощью as
from N import func as nfunc
mfunc(); nfunc()                # Вызов одного или другого
```

Расширение `as` функционирует в операторах `import` и `from` как инструмент переименования (его также можно применять для назначения короткого псевдонима длинному имени модуля в `import`); более подробно о такой форме речь пойдет в главе 25.

Пространства имен модулей

Модули вероятно лучше всего воспринимать как пакеты имен, т.е. места для определения имен, которые желательно сделать видимыми остальной части системы. Формально модули обычно соответствуют файлам, и Python создает объект модуля, содержащий все имена, которым выполнялось присваивание в файле модуля. Но говоря упрощенно, модули являются всего лишь пространствами имен (местами, где создаются имена), а существующие в модуле имена называются его *атрибутами*. В текущем разделе мы детально исследуем такую модель.

Файлы генерируют пространства имен

Ранее упоминалось, что файлы *преобразуются* в пространства имен, но как это на самом деле происходит? Короткий ответ: каждое имя, которому присваивается значение на верхнем уровне файла модуля (т.е. не внутри тела функции или класса), становится атрибутом данного модуля.

Например, при наличии оператора присваивания `X = 1` на верхнем уровне файла модуля `M.py` имя `X` становится атрибутом модуля `M`, на который мы можем ссылаться за пределами модуля как `M.X`. Имя `X` также становится глобальной переменной для остального кода внутри `M.py`, но для понимания причины нам необходимо чуть более формально рассмотреть понятие загрузки модулей и областей видимости.

- Операторы модуля выполняются при его первом импортировании. Когда модуль впервые импортируется в любом месте системы, Python создает пустой объект модуля и выполняет операторы из файла модуля друг за другом от начала до конца файла.
- Присваивания на верхнем уровне модуля создают его атрибуты. Во время выполнения операции импортирования не вложенные в def или class операторы на верхнем уровне файла, которые присваивают значения именам (например, =, def), создают атрибуты объекта модуля; присвоенные имена сохраняются в пространстве имен модуля.
- Доступ к пространству имен модуля можно получить через атрибут __dict__ или посредством dir(M). Пространства имен модулей, созданные операциями импортирования, представляют собой словари; к ним можно обращаться через встроенный атрибут __dict__, ассоциированный с объектами модулей, и инспектировать с помощью функции dir. Функция dir является приблизительным эквивалентом отсортированного списка ключей атрибута __dict__ объекта модуля, но содержит унаследованные имена для классов, может возвращать неполный список, а также предрасположена к изменениям от выпуска к выпуску.
- Модуль представляет собой одиночную область видимости (локальная является глобальной). Как выяснилось в главе 17, имена на верхнем уровне модуля следуют таким же правилам ссылки/присваивания, как имена в функции, но локальная и глобальная области видимости совпадают. Выражаясь более формально, они следуют правилу поиска в областях видимости LEGB, с которым мы встречались в главе 17, но без уровней просмотра L и E.

Тем не менее, важно понимать, что после того, как модуль был загружен, глобальная область видимости модуля становится словарем атрибутов *объекта* модуля. В отличие от области видимости функции, где локальное пространство имен существует только во время выполнения функции, область видимости файла модуля становится пространством имен атрибутов объекта модуля и продолжает существовать после окончания операции импортирования, предоставляя источник инструментов импортерам.

Ниже показана демонстрация описанных идей. Предположим, что мы создали в текстовом редакторе следующий файл модуля и назвали его module2.py:

```
print('starting to load...')      # начало загрузки...
import sys
name = 42

def func(): pass
class klass: pass

print('done loading.')           # загрузка окончена.
```

Когда этот модуль импортируется (или запускается как программа) в первый раз, Python выполняет его операторы от начала до конца. Одни операторы в качестве подобного эффекта создают имена в пространстве имен модуля, но другие делают фактическую работу, пока длится операция импортирования. Например, во время импортирования выполняются два оператора print из данного файла:

```
>>> import module2
starting to load...
done loading.
```

После того, как модуль загружен, его область видимости становится пространством имен атрибутов в объекте модуля, который мы получаем из оператора `import`. Затем мы можем получать доступ к атрибутам в данном пространстве имен, уточняя их именем включающего модуля:

```
>>> module2.sys
<module 'sys' (built-in)>
>>> module2.name
42
>>> module2.func
<function func at 0x000000000222E7B8>
>>> module2.klass
<class 'module2(klass)'>
```

Здесь `sys`, `name`, `func` и `klass` были присвоены значения во время выполнения операторов модуля, так что после импортирования все они являются атрибутами. Классы будут обсуждаться в части VI, но обратите внимание на атрибут `sys` – операторы `import` в действительности *присваивают* именам объекты модулей, и любое присваивание имени на верхнем уровне файла генерирует атрибут модуля.

Словари пространств имен: `__dict__`

На самом деле внутренне пространства имен модулей хранятся как *словарные* объекты. Они представляются собой нормальные словари со всеми обычными методами. Когда необходимо (скажем, при написании инструментов, которые выводят содержимое модулей обобщенным образом, что будет делаться в главе 25), мы можем получать доступ к словарю пространства имен модуля через атрибут `__dict__` объекта модуля. Продолжим пример из предыдущего раздела (в Python 3.X не забывайте добавить вызов `list` – атрибут `__dict__` является объектом представления и его содержимое может отличаться за пределами использованной здесь версии Python 3.7):

```
>>> list(module2.__dict__.keys())
['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__file__',
 '__cached__', '__builtins__', 'sys', 'name', 'func', 'klass']
```

Имена, присвоенные в файле модуля, внутренне становятся ключами словаря, так что некоторые имена отражают присваивания верхнего уровня в файле. Однако Python также автоматически добавляет в пространство имен модуля несколько дополнительных имен; например, `__file__` дает имя файла, из которого загружался модуль, а `__name__` – его имя, как оно известно импортерам (без расширения .ru и пути к каталогу). Чтобы посмотреть только имена, присвоенные в коде, отфильтруем имена с двумя подчеркиваниями, как поступали ранее при описании функции `dir` в главе 15 и встроенной области видимости в главе 17:

```
>>> list(name for name in module2.__dict__.keys() if not name.startswith('__'))
['sys', 'name', 'func', 'klass']
>>> list(name for name in module2.__dict__ if not name.startswith('__'))
['sys', 'name', 'func', 'klass']
```

На этот раз мы фильтруем с помощью *генератора*, а не спискового включения, и можем опустить `.keys()`, т.к. словари генерируют свои ключи автоматически, хотя и неявно; эффект будет таким же. Аналогичные словари `__dict__` мы еще встретим при рассмотрении в части VI объектов, основанных на *классах*. В обоих случаях извле-

чение атрибутов похоже на индексацию словарей, но в классах задействовано наследование:

```
>>> module2.name, module2.__dict__['name']
(42, 42)
```

Уточнение имен атрибутов

Говоря об извлечении атрибутов, поскольку вы уже лучше знакомы с модулями, настало время более формально закрепить понятие уточнения имен. В Python доступ к атрибутам любого объекта, имеющего атрибуты, производится с применением синтаксиса *уточнение объект. атрибут* (также известного как извлечение атрибутов).

Уточнение на самом деле представляет собой выражение, возвращающее значение, которое было присвоено имени атрибута, ассоциированного с объектом. Скажем, выражение `module2.sys` в предыдущем примере извлекает значение, присвоенное `sys` в `module2`. Подобным же образом, если есть встроенный списковый объект `L`, тогда `L.append` возвратит объект метода `append`, ассоциированный с этим списком.

Важно помнить о том, что уточнение атрибутов не имеет ничего общего с правилами поиска в областях видимости, рассмотренными в главе 17; оно является независимой концепцией. Когда уточнение используется для доступа к именам, Python дается явный объект, из которого необходимо извлечь указанные имена. Правило поиска в областях видимости LEGB применяется только к неуточненным именам — его можно использовать для крайнего слева имени в пути, но имена, находящиеся дальше после точек, взамен ищут специфические объекты. Ниже описаны принятые правила.

Простые переменные

`X` означает поиск имени `X` в текущих областях видимости (следуя правилу LEGB из главы 17).

Уточнение

`X.Y` означает поиск `X` в текущих областях видимости, затем поиск атрибута `Y` в объекте `X` (не в областях видимости).

Пути уточнения

`X.Y.Z` означает поиск имени `Y` в объекте `X`, затем поиск `Z` в объекте `X.Y`.

Всеобщность

Уточнение работает для всех объектов с атрибутами: модулей, классов, типов расширений С и т.д.

В части VI мы увидим, что уточнение атрибутов имеет чуть большее значение для классов (она также является местом, где действует то, что называется *наследованием*), но в целом кратко описанные здесь правила применяются ко всем именам в Python.

Импортирование или область видимости

Как мы выяснили, доступ к именам, определенным в другом файле модуля, невозможен без первоначального импортирования этого файла. То есть вы никогда автоматически не увидите имена в другом файле безотносительно структуры операций импортирования или вызовов функций в своей программе. Смысл переменной всегда определяется местоположениями присваиваний в исходном коде, а атрибуты всегда запрашиваются с явным указанием объекта.

Например, рассмотрим два следующих простых модуля. В первом модуле, `moda.py`, определяется переменная `X` как глобальная по отношению к коду только в данном файле наряду с функцией, которая изменяет глобальную переменную `X` в этом файле:

```
X = 88          # Переменная X: глобальная по отношению только к данному файлу
def f():
    global X      # Изменить X из этого файла
    X = 99        # Нельзя видеть имена из других модулей
```

Во втором модуле, `modb.py`, определяется собственная глобальная переменная `X`, а также импортируется и вызывается функция из первого модуля:

```
X = 11          # Переменная X: глобальная по отношению только к данному файлу
import moda      # Получить доступ к именам в moda
moda.f()         # Устанавливает moda.X, но не X из этого файла
print(X, moda.X)
```

Во время выполнения `moda.f` изменяет `X` в `moda`, но не `X` в `modb`. Глобальной областью видимости для функции `moda.f` всегда будет включающий ее файл вне зависимости от того, из какого модуля она в итоге вызывается:

```
% python modb.py
11 99
```

Другими словами, операции импортирования никогда не обеспечивают восходящую видимость коду в импортируемых файлах – импортируемый файл не может видеть имена из файла, в котором он импортируется. Более формально:

- функции не могут видеть имена в других функциях, если только они физически не вкладываются в них;
- код модуля не может видеть имена в других модулях, если только он явно не импортирует их.

Такое поведение является частью понятия *лексической области видимости* – в Python области видимости, окружающие порцию кода, полностью определяются физическим местоположением кода в файле. Области видимости не подвержены влиянию со стороны вызовов функций или операций импортирования модулей¹.

Вложение пространств имен

В определенном смысле, хотя операции импортирования не вкладывают пространства имен снизу вверх, они делают это сверху вниз. То есть, несмотря на то, что импортируемый модуль не имеет прямого доступа к именам в файле, который его импортирует, с использованием путей уточнения атрибутов становится возможным спуск внутрь произвольно глубоко вложенных модулей и доступ к их атрибутам. В качестве примера рассмотрим показанные далее три файла. В `mod3.py` определяется единственное глобальное имя и атрибут за счет присваивания:

```
X = 3
```

В свою очередь в `mod2.py` определяется собственное имя `X`, после чего импортируется `mod3` и применяется уточнение для доступа к атрибуту импортируемого модуля:

¹ Некоторые языки поступают по-другому и предлагают *динамические области видимости*, которые действительно могут зависеть от вызовов во время выполнения. Однако такой подход имеет тенденцию усложнять код, потому что смысл переменной может отличаться с течением времени. В Python области видимости более просто соответствуют тексту программы.

```
X = 2
import mod3

print(X, end=' ')      # Собственное глобальное имя X
print(mod3.X)          # X из mod3
```

В модуле `mod1.py` также определяется собственное имя `X`, затем импортируется `mod2` и извлекаются атрибуты из первого и второго модулей:

```
X = 1
import mod2

print(X, end=' ')      # Собственное глобальное имя X
print(mod2.X, end=' ') # X из mod2
print(mod2.mod3.X)     # X из вложенного mod3
```

На самом деле, когда `mod1` импортирует `mod2`, он устанавливает двухуровневое вложение пространств имен. За счет использования пути имен `mod2.mod3.X` он может спуститься к модулю `mod3`, который вложен в импортируемый модуль `mod2`. Совокупный эффект заключается в том, что `mod1` может видеть имена `X` во всех трех файлах и потому имеет доступ ко всем трем глобальным областям видимости:

```
% python mod1.py
2 3
1 2 3
```

Однако обратное утверждение неверно: `mod3` не может видеть имена в `mod2`, а `mod2` не может видеть имена в `mod1`. Возможно, пример будет легче понять, если вы не будете думать в терминах пространств имен и областей видимости, а взамен сконцентрируетесь на задействованных объектах. `mod2` внутри `mod1` – это всего лишь имя, которое ссылается на объект с атрибутами, часть которых может ссылаться на другие объекты с атрибутами (`import` является присваиванием). Для путей вроде `mod2.mod3.X` интерпретатор Python просто производит оценку слева направо, попутно извлекая атрибуты из объектов. Обратите внимание, что в `mod1` можно записать `import mod2` и затем `mod2.mod3.X`, но нельзя записать `import mod2.mod3` – такой синтаксис инициирует то, что называется импортированием *пакетов* (каталогов), описанное в следующей главе. Импортирование пакетов также создает вложение пространств имен модулей, но его операторы `import` служат для отражения деревьев каталогов, а не простых цепочек импортирования файлов.

Перезагрузка модулей

Как объяснялось ранее, по умолчанию код модуля выполняется только один раз на процесс. Чтобы принудительно перезагрузить и повторно выполнить код модуля, понадобится явно запросить у Python такие действия, вызвав встроенную функцию `reload`. В этом разделе мы исследуем, как с помощью перезагрузок делать свои системы более динамичными. Вот краткое изложение.

- Операции импортирования (посредством операторов `import` и `from`) загружают и выполняют код модуля только при первом импортировании модуля в процессе.
- Последующие операции импортирования применяют объект уже загруженного модуля, не перезагружая и не выполняя повторно код из файла модуля.
- Функция `reload` вынуждает код уже загруженного модуля перезагрузиться и повторно выполниться. Присваивания в новом коде файла изменяют существующий объект модуля на месте.

Зачем заботиться о перезагрузке модулей? Если вкратце, то для *динамической настройки*: функция `reload` делает возможным изменение частей программы без остановки всей программы. Функция `reload` позволяет незамедлительно наблюдать эффекты от изменений в компонентах. Перезагрузка не помогает абсолютно в каждой ситуации, но там, где это происходит, она значительно сокращает цикл разработки. Например, представим себе программу базы данных, которая должна подключаться к серверу при начальном запуске; из-за того, что изменения или настройки в программе могут тестироваться сразу же после перезагрузок, во время отладки придется подключаться только один раз. Длительно выполняющиеся серверы тоже могут обновлять себя таким способом.

Поскольку язык Python является интерпретируемым (более или менее), он уже избавляется от шагов компиляции/связывания, через которые необходимо проходить, чтобы обеспечить запуск программы на С: модули загружаются динамически, когда импортируются выполняющейся программой. Перезагрузка предлагает добавочное преимущество в плане производительности, позволяя также изменять части функционирующих программ без их остановки.

Хотя данная тема выходит за рамки книги, имейте в виду, что функция `reload` в текущий момент работает только с модулями, написанными на Python. Скомпилированные модули расширений, реализованные на языках вроде С, также могут динамически загружаться во время выполнения, но перезагружать их нельзя (правда, большинство пользователей, вероятно, предпочтут писать настройки на Python).



Примечание, касающееся нестыковки версий. В Python 2.X инструмент `reload` доступен в виде встроенной функции. В Python 3.X он был перемещен в стандартный библиотечный модуль и доступен как `imp.reload`. Это просто означает, что в Python 3.X требуется дополнительный оператор `import` или `from` для загрузки данного инструмента. Читатели, использующие Python 2.X, могут игнорировать такие операции импортирования в примерах книги или применять в любом случае – Python 2.X также имеет `reload` в своем модуле `imp` для облегчения миграции в Python 3.X. Перезагрузка работает одинаково вне зависимости от пакетирования.

Основы использования `reload`

В отличие от `import` и `from`:

- `reload` является функцией в Python, а не оператором;
- `reload` передается существующий объект модуля, а не новое имя;
- `reload` находится внутри модуля в Python 3.X и сама должна быть импортирована.

Из-за того, что `reload` ожидает объект, модуль должен быть предварительно успешно загружен, прежде чем его можно будет перезагружать (если операция импортирования оказалась безуспешной по причине синтаксической или другой ошибки, тогда может понадобиться повторить ее, чтобы появилась возможность перезагрузки модуля). Кроме того, синтаксис операторов `import` и вызовов `reload` различается: как вызовы функции перезагрузки требуют круглых скобок, но операторы импортирования – нет. Абстрактно перезагрузка выглядит примерно так:

```
import module          # Первоначальная операция импортирования
...использовать атрибуты module...
...                      # Внести изменения в файл модуля
```

```
...
from imp import reload          # Получить саму функцию reload (в Python 3.x)
reload(module)                 # Получить обновленные атрибуты
...использовать атрибуты module...
```

Типичная модель использования предусматривает импортирование модуля, изменение его исходного кода в текстовом редакторе и затем перезагрузку. Указанные действия могут происходить при работе в интерактивном сеансе, но также и в более крупных программах, которые требуют периодической перезагрузки.

Когда вызвана функция `reload`, Python повторно читает исходный код файла модуля и запускает его операторы верхнего уровня. Возможно, самая важная особенность функции `reload` заключается в том, что она изменяет объект модуля *на месте*, не удаляя и не создавая его заново. По этой причине перезагрузка автоматически оказывает влияние на каждую ссылку на полный *объект* модуля повсюду в программе. Ниже описаны детали.

- Функция `reload` запускает новый код файла модуля в текущем пространстве имен модуля. Повторное выполнение кода файла модуля переписывает его существующее пространство имен, а не удаляет и воссоздает его.
- Присваивания верхнего уровня в файле заменяют имена новыми значениями. Скажем, повторное выполнение оператора `def` заменяет предыдущую версию функции в пространстве имен модуля, делая новое присваивание имени функции.
- Перезагрузки воздействуют на всех клиентов, которые применяют `import` для извлечения модулей. Поскольку клиенты, которые используют `import`, производят уточнение для извлечения атрибутов, после перезагрузки они обнаружат новые значения в объекте модуля.
- Перезагрузки воздействуют только на клиентов, которые применяют `from` в будущем. Перезагрузка не окажет влияние на клиентов, которые использовали `from` для извлечения атрибутов в прошлом; они по-прежнему будут иметь ссылки на старые объекты, извлеченные перед перезагрузкой.
- Перезагрузки применяются только к одиночному модулю. Вы должны запускать их для каждого модуля, который желаете обновить, либо воспользоваться кодом или инструментами, последовательно применяющими перезагрузки.

Пример использования `reload`

В целях демонстрации далее представлен конкретный пример `reload` в действии. Мы изменим и перезагрузим файл модуля, не останавливая интерактивный сеанс Python. Перезагрузки используются также во многих других сценариях (как описано во врезке “Что потребует внимания: перезагрузка модулей” далее в главе), но ради иллюстрации мы будем сохранять вещи простыми. С помощью выбранного текстового редактора создайте файл модуля по имени `changer.py` со следующим содержимым:

```
message = "First version"      # Первая версия
def printer():
    print(message)
```

В модуле создаются и экспортируются два имени — одно привязано к строке, а другое к функции. Теперь запустите интерпретатор Python, импортируйте модуль и вызовите функцию, которую он экспортирует. Функция выведет значение глобальной переменной `message`:

```
% python  
>>> import changer  
>>> changer.printer()  
First version
```

Оставив интерпретатор активным, отредактируйте файл модуля в другом окне:

```
...модифицировать changer.py, не останавливая Python...  
% notepad changer.py
```

Измените глобальную переменную `message`, а также тело функции `printer`:

```
message = "After editing"          # После редактирования  
def printer():  
    print('reloaded:', message)    # перезагружен
```

Затем возвратитесь в окно сеанса Python и перезагрузите модуль, чтобы извлечь новый код. В показанном ниже взаимодействии обратите внимание на то, что повторное импортирование модуля никакого воздействия не оказывает; мы получаем первоначальное сообщение, хотя файл был изменен. Для получения новой версии потребуется вызвать `reload`:

```
...возвратиться в интерпретатор Python...  
>>> import changer  
>>> changer.printer()          # Безрезультатно: используется загруженный модуль  
First version  
>>> from imp import reload  
>>> reload(changer)           # Заставляет новый код загрузиться/выполниться  
<module 'changer' from '.\\changer.py'>  
>>> changer.printer()         # Теперь выполняется новая версия  
reloaded: After editing
```

Следует отметить, что функция `reload` на самом деле *возвращает* объект модуля — ее результат обычно игнорируется, но поскольку результаты выражений выводятся в интерактивной подсказке, Python отображает стандартное представление `<module 'имя' ...>`.

Осталось сделать два финальных замечания. Во-первых, если вы применяете функцию `reload`, то возможно захотите сочетать ее с оператором `import`, а не `from`, т.к. последний не обновляется операциями перезагрузки, оставляя имена в довольно странном состоянии (см. раздел “Затруднения, связанные с модулями” в главе 25). Во-вторых, функция `reload` сама по себе обновляет только *одиночный* модуль, но легко написать функцию, которая последовательно применяет `reload` к связанным модулям — расширение, рассматриваемое в учебном примере в конце главы 25.

Что потребует внимания: перезагрузка модулей

Помимо возможности перезагружать (а потому повторно выполнять) модули в интерактивной подсказке перезагрузка модулей также полезна в более крупных системах, особенно когда затраты на перезапуск целого приложения непомерно высоки. Например, главными кандидатами для динамической перезагрузки могут служить игровые серверы и системы, которые должны подключаться через сеть при начальном запуске.

Они также полезны в работе графического пользовательского интерфейса (действие обратного вызова какого-то виджета можно изменять, оставляя графический пользовательский интерфейс активным) и в ситуациях, когда Python используется в качестве встроенного языка в программе на C или C++ (включающая программа

может запрашивать перезагрузку выполняемого ею кода Python без необходимости в остановке). Дополнительные сведения о перезагрузке обратных вызовов графического пользовательского интерфейса и встроенного кода Python смите в книге *Programming Python* (<http://www.oreilly.com/catalog/9780596158101>).

В более общем плане перезагрузка дает возможность программам предоставлять крайне динамичные интерфейсы. Скажем, Python часто применяется как язык *настройки* для более крупных систем – пользователи могут настраивать продукты за счет написания небольшого объема кода Python на месте эксплуатации, не требуя перекомпиляции всего продукта (или даже вообще не имея дел с его исходным кодом). В таких мирах код Python уже сам по себе привносит изюминку динамичности.

Однако чтобы быть еще более динамичными, такие системы могут автоматически перезагружать код настройки Python с определенной периодичностью во время выполнения. В итоге изменения со стороны пользователей подхватываются во время функционирования системы; нет нужды останавливать и перезапускать ее при каждой модификации кода Python. Не все системы требуют подобного подхода к динамизму, но для тех, которым он необходим, перезагрузка модулей предоставляет легкий в использовании инструмент динамической настройки.

Резюме

В главе рассматривались важнейшие основы инструментов для написания кода модулей – операторы `import` и `from`, а также функция `reload`. Вы узнали, что оператор `from` просто добавляет дополнительный шаг, который копирует имена из файла после того, как он был импортирован, и что `reload` заставляет файл импортироваться повторно без остановки и перезапуска Python. Вы также ознакомились с концепциями пространств имен, выяснили, что происходит, когда операции импортирования являются вложенными, изучили способ, которым файлы становятся пространствами имен модулей, и узнали о нескольких потенциальных затруднениях, связанных с оператором `from`.

Хотя вы уже знаете достаточно много для того, чтобы применять файлы модулей в своих программах, в следующей главе обзор модели импортирования будет расширен представлением *операций импортирования пакетов* – способа указания в операторах `import` части пути к каталогу с желаемым модулем. Вы увидите, что операции импортирования пакетов дают иерархию, которая полезна в более крупных системах и позволяет устраниить конфликты между одинаково именованными модулями. Тем не менее, прежде чем двигаться дальше, закрепите пройденный материал этой главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Каким образом вы будете создавать модуль?
2. Как оператор `from` связан с оператором `import`?
3. Как функция `reload` связана с операциями импортирования?
4. Когда вы обязаны использовать `import` вместо `from`?
5. Назовите три потенциальных затруднения, связанных с оператором `from`.
6. Какова... скорость полета необремененной грузом ласточки?

Проверьте свои знания: ответы

- Чтобы создать модуль, вы просто помещаете операторы Python в текстовый файл; каждый файл исходного кода автоматически считается модулем и не существует какого-то синтаксиса для его объявления. Операции импортирования загружают файлы модулей внутрь объектов модулей, расположенных в памяти. Создать модуль можно также за счет написания кода на внешнем языке, подобном C или Java, но такие модули расширений в книге не рассматриваются.
- Оператор `from` импортирует целый модуль, как и оператор `import`, но в качестве добавочного шага также копирует одну или несколько переменных из импортируемого модуля в область видимости, где находится `from`. Это позволяет использовать импортированные имена напрямую (`имя`) вместо того, чтобы дополнять их именем модуля (`модуль.имя`).
- По умолчанию модуль импортируется только один раз на процесс. Функция `reload` заставляет модуль импортироваться повторно. Она применяется главным образом для подхвата новых версий исходного кода модуля на стадии разработки и в сценариях динамической настройки.
- Вы обязаны использовать `import` вместо `from`, только когда нуждаетесь в доступе к одному и тому же имени из двух разных модулей; поскольку вам придется указать имена включающих модулей, два имени будут уникальными. Расширение `as` также может сделать `from` пригодным для употребления в таком контексте.
- Оператор `from` способен затенить смысл переменной (в каком модуле она определена), иметь проблемы с вызовом `reload` (имена могут ссылаться на предыдущие версии объектов) и искажать пространства имен (молча переписывать имена, используемые в вашей области видимости). Форма `from *` хуже в еще больших отношениях – она может серьезно искажать пространства имен и делать неясным смысл переменных, поэтому вероятно ее лучше применять умеренно.
- Какую ласточку вы имеете в виду – африканскую или европейскую?

Пакеты модулей

До сих пор при импортировании модулей мы загружали файлы. Такая методика отражает типичное использование модулей и обычно применяется для большинства операций импортирования в начале карьеры программиста на Python. Однако история импортирования модулей несколько интереснее, чем можно было предположить.

При импортировании в дополнение к имени модуля можно указывать путь к каталогу. Каталог с кодом Python называют *пакетом*, поэтому операции импортирования подобного рода известны как *импортирование пакетов*. В сущности, импортирование пакетов превращает каталог на диске вашего компьютера в еще одно пространство имен Python с атрибутами, соответствующими подкаталогам и файлам модулей, которые содержит каталог.

Хотя эта возможность несколько сложнее обычного импортирования, обеспечиваемая ею иерархия может оказаться удобной для организации файлов в крупной системе и способствует упрощению настройки пути поиска модулей. Как будет показано, импортирование пакетов иногда требуется для устранения неоднозначностей импортирования, когда на компьютере установлено много программных файлов с одинаковыми именами.

Мы также рассмотрим здесь недавно появившуюся в Python модель *относительного импортирования* и ее синтаксис, поскольку она касается только кода в пакетах. Упомянутая модель модифицирует пути поиска в Python 3.X и расширяет оператор `from` для операций импортирования внутри пакетов в Python 2.X и 3.X. Эта модель может сделать такие внутривидовые операции импортирования более явными и лаконичными, но сопровождается рядом компромиссов, которые способны повлиять на ваши программы.

Наконец, для читателей, использующих Python 3.3 и последующие версии, в главе будет представлена новая модель *пакетов пространств имен*, которая позволяет пакетам охватывать множество каталогов и не требовать инициализационных файлов. Такая модель пакетов нового стиля необязательна и может применяться во взаимодействии с первоначальной (теперь называемой “обычной”) моделью пакетов, но она отвергает некоторые основные идеи и правила первоначальной модели. По указанной причине сначала будут исследоваться обычные пакеты, что предназначено для всех читателей, а затем пакеты пространств имен в качестве необязательной темы.

Основы импортирования пакетов

На элементарном уровне операции импортирования пакетов прямолинейны — там, где в операторах `import` находилось имя простого файла, взамен можно указать *путь с именами, разделенными точками*:

```
import dir1.dir2.mod
```

То же самое касается операторов `from`:

```
from dir1.dir2.mod import x
```

Предполагается, что “точечный” путь в таких операторах соответствует пути в иерархии каталогов на диске к файлу `mod.py` (или подобному; расширение может варьироваться). То есть предшествующие операторы указывают на то, что диск вашего компьютера содержит каталог `dir1`, который имеет подкаталог `dir2`, содержащий файл модуля `mod.py` (или подобный).

Кроме того, показанные выше операции импортирования подразумевают, что `dir1` находится внутри контейнерного каталога `dir0`, который является компонентом нормального пути поиска модулей Python. Другими словами, эти два оператора `import` означают наличие структуры каталогов следующего вида (приведенной с разделителями в форме обратной косой черты, принятой в Windows):

```
dir0\dir1\dir2\mod.py    # Или mod.py, mod.so и т.д.
```

Контейнерный каталог `dir0` должен быть добавлен в путь поиска модулей при условии, что он не представляет собой домашний каталог файла верхнего уровня, в частности как если бы имя `dir1` обозначало простой файл модуля.

Выражаясь более формально, крайний слева компонент в пути операции импортирования пакета по-прежнему является *относительным* к какому-то каталогу, включенному в список пути поиска модулей `sys.path`, который обсуждался в главе 22. Тем не менее, начиная с этого каталога, операторы импортирования в сценарии явно указывают пути, ведущие к модулям в пакетах.

Пакеты и настройки пути поиска

Если вы используете данное средство, тогда имейте в виду, что пути к каталогам в операторах импортирования могут быть только *переменными, разделенными точками*. В операторах импортирования не разрешено применять любой синтаксис, специфичный для платформы, такой как `C:\dir1, My Documents.dir2` или `../dir1`. Взамен используйте синтаксис, специфичный к платформе, в настройках пути поиска для указания имен контейнерных каталогов.

Например, в приведенных ранее операторах импортирования `dir0` (имя каталога, добавленное в путь поиска модулей), может быть произвольно длинным и специфичным к платформе путем к каталогу, содержащему `dir1`. Нельзя применять недопустимый оператор вроде:

```
import C:\mycode\dir1\dir2\mod    # Ошибка: недопустимый синтаксис
```

Но можно добавить `C:\mycode` в переменную среды `PYTHONPATH` или в файл `.pth` и затем поместить в сценарий такой оператор:

```
import dir1.dir2.mod
```

В действительности элементы в пути поиска модулей предоставляют специфичные для платформы *префиксы* путей, которые ведут к крайним слева именам в операторах

`import` и `from`. Сами операторы импортирования определяют остаток пути к каталогу в независимом от платформы стиле¹.

Что касается простых операций импортирования файлов, то вам не нужно добавлять контейнерный каталог `dir0` в путь поиска модулей, когда он уже в нем присутствует. Как было указано в главе 22, он там будет находиться, если является домашним каталогом файла верхнего уровня, каталогом, где вы работаете интерактивно, каталогом стандартной библиотеки или подкаталогом `site-packages` с установленными сторонними расширениями. Однако, так или иначе, путь поиска модулей обязан включать все каталоги, содержащие крайние слева компоненты из операторов импортирования пакетов в коде.

Файлы `__init__.py` пакетов

Если вы решили использовать импортирование пакетов, то вам придется соблюдать еще одно ограничение: по крайней мере, до версии Python 3.3 каждый каталог, указанный внутри пути в операторе импортирования пакета, должен содержать файл по имени `__init__.py`, иначе операция импортирования пакета потерпит неудачу. То есть в рассмотренных примерах каталоги `dir1` и `dir2` обязаны содержать файл `__init__.py`; контейнерному каталогу `dir0` такой файл не требуется, потому что сам он в операторе `import` не указан.

Более формально для структуры каталогов следующего вида:

```
dir0\dir1\dir2\mod.py
```

и оператора `import` в форме:

```
import dir1.dir2.mod
```

применимы перечисленные ниже правила.

- Каталоги `dir1` и `dir2` должны содержать файл `__init__.py`.
- Каталог `dir0`, контейнер, не требует файла `__init__.py`; если этот файл существует, то он просто будет проигнорирован.
- Каталог `dir0`, но не `dir0\dir1`, должен находиться в пути поиска модулей `sys.path`.

Для соблюдения первых двух правил создатели пакетов должны обеспечить наличие файлов исследуемого здесь вида. Для соблюдения последнего правила каталог `dir0` обязан быть автоматическим компонентом пути поиска (домашним каталогом, каталогом стандартной библиотеки или каталогом внутри `site-packages`) либо присутствовать в переменной `PYTHONPATH`, в настройках файла `.pth` или во вносимых вручную изменениях `sys.path`.

¹ Синтаксис путей с точками был выбран отчасти для обеспечения нейтральности к платформе, но еще и потому, что пути в операторах `import` становятся реальными цепочками вложенных объектов. Такой синтаксис также означает, вы можете получать странные сообщения об ошибках, если забудете опустить `.py` в своих операторах `import`. Скажем, оператор `import mod.py` предположительно является операцией импортирования пути к каталогу — он загружает `mod.py`, затем пытается загрузить `mod\py.py` и в конечном итоге выдает потенциально сбивающее с толку сообщение об ошибке “No module named `py`” (“Модуль по имени `py` отсутствует”). Начиная с версии Python 3.3, сообщение об ошибке было улучшено и выглядит как “No module named ‘mod.py’; mod is not a package” (“Модуль по имени `mod.py` отсутствует; `mod` не является пакетом”).

В результате структура каталогов для данного примера должна выглядеть так (отступы обозначают вложение каталогов):

```
dir0\                                # Контейнер в пути поиска модулей
  dir1\
    __init__.py
  dir2\
    __init__.py
    mod.py
```

Файлы `__init__.py` могут содержать код Python подобно нормальным файлам модулей. Они имеют специальные имена, поскольку их код выполняется автоматически, когда Python импортирует каталог в первый раз, и потому служат главным образом в качестве привязок для выполнения шагов инициализации, требующихся пакету. Тем не менее, такие файлы могут также быть пустыми и временами исполнять дополнительные роли, как объясняется в следующем разделе.



Как будет показано ближе к концу главы, в Python 3.3 и последующих версиях требование наличия файлов по имени `__init__.py` в пакетах было отменено. В версии Python 3.3 и далее каталоги модулей без таких файлов могут импортироваться как *пакеты пространств имен* с единственным каталогом, которые работают точно так же, но не запускают файл кода, относящийся к стадии инициализации. Однако в версиях, предшествующих Python 3.3, и во всех версиях Python 2.X пакеты по-прежнему требуют наличия файлов `__init__.py`. Как будет описано далее, в Python 3.3 и последующих версиях эти файлы, когда используются, также обеспечивают выигрыш в производительности.

Роли инициализационных файлов пакетов

Выражаясь более подробно, файл `__init__.py` служит привязкой для действий стадии инициализации пакета, объявляет каталог пакетом Python, генерирует пространство имен для каталога и реализует поведение операторов `from *` (т.е. `from .. import *`) в случае применения с операциями импортирования каталогов.

Инициализация пакетов

Когда программа Python импортирует каталог в первый раз, производится автоматический запуск всего кода из файла `__init__.py` каталога. По этой причине такие файлы являются естественным местом для размещения кода, который инициализирует состояние, требующееся файлам в пакете. Например, пакет может использовать свой инициализационный файл для создания обязательных файлов данных, установления подключений к базам данных и т.д. Обычно файлы `__init__.py` не планируются быть полезными при выполнении напрямую; они запускаются автоматически, когда впервые производится доступ к пакету.

Объявления применимости модулей

Файлы `__init__.py` пакетов также в некоторой степени присутствуют для объявления о том, что каталог является пакетом Python. В данной роли эти файлы помогают не допустить, чтобы каталоги с распространенными именами непреднамеренно скрывали настоящие модули, которые обнаруживаются позже в пути поиска модулей. Без такой меры предосторожности Python мог бы выбрать ка-

талог, который не имеет никакого отношения к вашему коду, просто потому, что он оказался вложенным в каталог, находящийся раньше в пути поиска. Как мы увидим позже, пакеты пространств имен, появившиеся в Python 3.3, позволяют в основном избавиться от этой роли, но достигают похожего эффекта алгоритмически за счет просмотра вперед пути для поиска более поздних файлов.

Инициализация пространства имен модуля

В модели импортирования пакетов пути к каталогам в сценарии после импорта становятся реальными цепочками вложенных объектов. Скажем, в предыдущем примере после импортирования выражение `dir1.dir2` работает и возвращает объект модуля, который содержит все имена, присвоенные инициализационным файлом `__init__.py` в каталоге `dir2`. Такие файлы предоставляют пространство имен для объектов модулей, созданных на основе каталогов, которые иначе не имели бы реального ассоциированного файла модуля.

Поведение оператора `from *`

В качестве расширенной возможности вы можете использовать в файлах `__init__.py` списки `__all__` для определения, что экспортить, когда каталог импортируется с помощью формы оператора `from *`. В файле `__init__.py` список `__all__` представляет собой перечень имен подмодулей, которые должны автоматически импортироваться в случае применения `from *` для имени пакета (каталога). Если список `__all__` не установлен, тогда оператор `from *` не загружает автоматически подмодули, вложенные в каталог; взамен он загружает только имена, которые определены присваиваниями в файле `__init__.py` каталога, в том числе любые подмодули, явно импортируемые кодом в данном файле. Например, оператор `from подмодуль import X` в файле `__init__.py` каталога делает имя `X` доступным в пространстве имен этого каталога. (В главе 25 будут показаны дополнительные роли для списка `__all__`: он обеспечивает объявление экспортации `from *` простых файлов.)

Вы также можете оставить файлы `__init__.py` пустыми, если их роли выходят за рамки имеющихся потребностей (и откровенно говоря, на практике они часто пусты). Тем не менее, файлы `__init__.py` должны существовать, чтобы операции импортирования каталогов вообще работали.



Не путайте файлы `__init__.py` пакетов и методы `__init__` конструкторов классов, с которыми мы встретимся в следующей части книги. Первые являются файлами кода, который выполняется, когда операции импортирования проходят через каталог пакета в первый раз во время выполнения программы, тогда как вторые вызываются при создании экземпляра. Оба средства исполняют роли, связанные с инициализацией, но в остальном они совершенно разные.

Пример импортирования пакетов

Давайте рассмотрим пример, который продемонстрирует в работе инициализационные файлы и пути. Следующие три файла находятся в каталоге `dir1` и его подкаталоге `dir2` – пути к ним указаны в комментариях:

```

# dir1\__init__.py
print('dir1 init')
x = 1
# dir1\dir2\__init__.py
print('dir2 init')
y = 2
# dir1\dir2\mod.py
print('in mod.py')
z = 3

```

Здесь `dir1` будет либо непосредственным подкаталогом каталога, в котором вы работаете (т.е. домашнего каталога), либо непосредственным подкаталогом каталога, присутствующего в пути поиска модулей (формально в `sys.path`). В любом случае контейнер `dir1` не нуждается в файле `__init__.py`.

Операторы `import` запускают инициализационный файл в каталоге при его первом обходе, когда Python спускается по пути; операторы `print` предусмотрены для отслеживания их выполнения:

```

C:\code> python          # Запускается в контейнерном каталоге dir1
>>> import dir1.dir2.mod # Первая операция импортирования выполняет
                           # инициализационные файлы
dir1 init
dir2 init
in mod.py
>>>
>>> import dir1.dir2.mod # Последующие операции импортирования не выполняют

```

Как и файлы модулей, уже импортированный каталог может передаваться функции `reload` для повторного выполнения инициализационных файлов. Ниже видно, что `reload` принимает точечное имя пути для перезагрузки вложенных каталогов и файлов:

```

>>> from imp import reload  # Оператор from необходим только в Python 3.X
>>> reload(dir1)
dir1 init
<module 'dir1' from '.\\dir1\\__init__.py'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from '.\\dir1\\dir2\\__init__.py'>

```

После импортирования путь в операторе `import` становится *цепочкой вложенных объектов* внутри сценария. Здесь `mod` – объект, вложенный в объект `dir2`, который в свою очередь вложен в объект `dir1`:

```

>>> dir1
<module 'dir1' from '.\\dir1\\__init__.py'>
>>> dir1.dir2
<module 'dir1.dir2' from '.\\dir1\\dir2\\__init__.py'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from '.\\dir1\\dir2\\mod.py'>

```

Фактически каждое имя каталога в пути делается переменной, которой присваивается объект модуля, чье пространство имен инициализируется всеми присваиваниеми в файле `__init__.py` этого каталога. Так, `dir1.x` ссылается на переменную `x`, присвоенную в `dir1__init__.py`, почти как `mod.z` ссылается на переменную `z`, присвоенную в `mod.py`:

```
>>> dir1.x  
1  
>>> dir1.dir2.y  
2  
>>> dir1.dir2.mod.z  
3
```

Использование `from` или `import` с пакетами

Возможно, операторы `import` несколько неудобно применять с пакетами, поскольку может требоваться частый набор путей в программе. Скажем, в примере из предыдущего раздела для доступа к `z` приходилось заново набирать и запускать полный путь от `dir1`. Если вы попробуете обратиться к `dir2` или `mod` напрямую, то получите ошибку:

```
>>> dir2.mod  
NameError: name 'dir2' is not defined  
Ошибка в имени: имя dir2 не определено  
>>> mod.z  
NameError: name 'mod' is not defined  
Ошибка в имени: имя mod не определено
```

По этой причине с пакетами часто удобнее использовать оператор `from`, чтобы избегать повторного набора путей при каждом доступе. Вероятно еще важнее то, что если вы когда-либо реструктурируете свое дерево каталогов, то оператор `from` потребует только одного обновления пути в коде, тогда как для операторов `import` могут понадобиться многочисленные обновления. Расширение `import as`, формально обсуждаемое в следующей главе, также способно помочь, предоставляя более короткий синоним для полного пути и инструмент переименования, когда то же самое имя появляется в нескольких модулях:

```
C:\code> python  
>>> from dir1.dir2 import mod          # Путь записывается только здесь  
dir1 init  
dir2 init  
in mod.py  
>>> mod.z                            # Повторять путь не нужно  
3  
>>> from dir1.dir2.mod import z  
>>> z  
3  
>>> import dir1.dir2.mod as mod      # Использовать более короткое имя  
# (см. главу 25)  
>>> mod.z  
3  
>>> from dir1.dir2.mod import z as modz # То же самое, если имена конфликтуют  
# (см. главу 25)  
>>> modz  
3
```

Для чего используется импортирование пакетов?

Если вы являетесь новичком в Python, тогда сначала должным образом освойте простые модули и только затем переходите к пакетам, которые представляют собой более сложное средство. Однако пакеты исполняют полезные роли особенно в крупных программах: они делают операции импортирования более информативными, служат организационным инструментом, упрощают путь поиска модулей и способны устранять неоднозначности.

Прежде всего, поскольку операции импортирования пакетов дают определенную информацию о каталогах в программных файлах, они облегчают нахождение файлов и выступают в качестве организационного инструмента.

Без путей пакетов для нахождения файлов часто приходится прибегать к пути поиска модулей. Кроме того, если вы организовали свои файлы в подкаталоги по областям функциональности, тогда операции импортирования пакетов сделают более очевидной роль, исполняемую тем или иным модулем, что в итоге улучшит читабельность кода. Например, нормальная операция импортирования файла из каталога где-то в пути поиска модулей вроде показанной ниже:

```
import utilities
```

предоставляет намного меньше информации, чем операция импортирования, которая включает путь:

```
import database.client.utilities
```

Импортирование пакетов может также значительно упростить настройки пути поиска в переменной PYTHONPATH и файлах .pth. Фактически если вы применяете явные операции импортирования пакетов для всех случаев импорта между каталогами и делаете их относительными к общему корневому каталогу, где хранится весь ваш код Python, то в пути поиска потребуется единственный элемент: общий корень. Наконец, импортирование пакетов помогает устранивать неоднозначности за счет точного указания, какие файлы нужно импортировать, а также избегать возникновения конфликтов, когда одно и то же имя модуля появляется в нескольких местах. Эта роль более детально исследуется в следующем разделе.

История о трех системах

Единственная ситуация, когда импортирование пакетов действительно является обязательным, связана с необходимостью устранения неоднозначностей, которые могут возникать в случае установки на одном компьютере множества программ с одинаково именованными файлами. В некоторой мере это проблема установки, но она может стать вопросом, требующим решения, и в общей практике – особенно учитывая стремление разработчиков использовать простые и похожие имена для файлов модулей. Давайте в целях иллюстрации рассмотрим один гипотетический сценарий.

Предположим, что программист разрабатывает программу на Python, которая содержит файл по имени utilities.py для общего служебного кода и файл верхнего уровня по имени main.py, запускаемый пользователями для старта программы. Для загрузки и работы с общим кодом в файлах программы применяется оператор import utilities. Программа поставляется в виде единственного файла .tar или .zip и при установке происходит распаковка всех файлов в одиночный каталог по имени system1 на целевом компьютере:

```
system1\  
    utilities.py      # Общие служебные функции и классы  
    main.py          # Файл, запускаемый для старта программы  
    other.py         # Импортирует utilities для загрузки инструментов
```

А теперь представим, что второй программист разрабатывает другую программу с файлами, тоже имеющими имена `utilities.py` и `main.py`, где снова повсюду используется оператор `import utilities` для загрузки общего файла кода. Когда вторая система извлекается и устанавливается на тот же компьютер, где находится первая система, ее файлы распаковываются в новый каталог по имени `system2`, гарантируя тем самым, что файлы с одинаковыми именами из первой системы не будут перезаписаны:

```
system2\  
    utilities.py      # Общие служебные инструменты  
    main.py          # Файл, запускаемый для старта программы  
    other.py         # Импортирует utilities
```

До сих пор проблем не было: обе системы могут сосуществовать и запускаться на одном и том же компьютере. На самом деле, чтобы работать с этими программами на своем компьютере, вам даже не придется конфигурировать путь поиска модулей. Поскольку Python всегда ищет сначала в домашнем каталоге (т.е. каталоге, содержащем файл верхнего уровня), операции импортирования в файлах любой из двух систем автоматически будут видеть все файлы в каталоге этой системы. Скажем, если вы запустите `system1\main.py`, то все операции импортирования будут производить поиск сначала в каталоге `system1`. Аналогично при запуске `system2\main.py` поиск будет осуществляться сначала в каталоге `system2`. Вспомните, что настройки пути поиска модулей нужны только для импортирования между границами каталогов.

Тем не менее, пусть после установки указанных двух программ на своем компьютере вы решили задействовать определенный код из каждого файла `utilities.py` в собственной системе. В конце концов, он является общим служебным кодом, а код Python по своей натуре “желает” быть многократно используемым. В таком случае вы хотели бы иметь возможность применять в своем коде, который будет помещен в третий каталог, приведенные ниже операторы для загрузки одного из двух файлов `utilities.py`:

```
import utilities  
utilities.func('spam')
```

И вот теперь проблема начинает вырисовываться. Чтобы код вообще заработал, вам придется установить путь поиска модулей, включив в него каталоги, которые содержат файлы `utilities.py`. Но какой каталог вы укажете первым в пути – `system1` или `system2`?

Проблемой является линейная природа пути поиска. Он всегда просматривается слева направо, поэтому независимо от того, сколько времени вы размышляете над данной дилеммой, вы всегда будете получать только один файл `utilities.py` – из каталога, который указан первым (крайний слева) в пути поиска. В том виде, как есть, вы никогда не сможете импортировать файл `utilities.py` из другого каталога.

Вы могли бы попробовать изменить `sys.path` внутри своего сценария перед каждой операцией импортирования, но такой подход сопряжен с добавочной работой и подвержен ошибкам. Изменение переменной среды `PYTHONPATH` перед запуском каждой программы на Python слишком утомительно и не позволяет использовать обе версии в одном файле одновременно. По умолчанию вы находитесь в тупике.

В действительности проблему решают пакеты. Вместо установки программ в независимые каталоги, индивидуально перечисленные в пути поиска модулей, вы можете пакетировать и устанавливать их как *подкаталоги* под общим корнем. Вот как можно было бы организовать весь код в текущем примере:

```
root\
    system1\
        __init__.py
        utilities.py
        main.py
        other.py
    system2\
        __init__.py
        utilities.py
        main.py
        other.py
    system3\          # Здесь или где-то в другом месте
        __init__.py      # Файл __init__.py требуется только в случае
                          # импортирования где-то еще
        myfile.py        # Содержит ваш новый код
```

Теперь добавьте общий корневой каталог в путь поиска. Если все операции импортирования в вашем коде являются относительными к общему корню, тогда вы можете импортировать файл `utilities.py` любой из двух систем с помощью операции импортирования пакета – имя включающего каталога делает путь (и, следовательно, ссылку на модуль) уникальным. На самом деле вы можете импортировать в том же самом модуле *оба* файла `utilities.py` при условии, что применяете оператор `import` и повторяете полный путь каждый раз, когда ссылаетесь на служебные модули:

```
import system1.utilities
import system2.utilities
system1.utilities.function('spam')
system2.utilities.function('eggs')
```

Имена включающих каталогов делают ссылки на модули уникальными.

Имейте в виду, что вы должны использовать с пакетами оператор `import` вместо `from`, только если нуждаетесь в доступе к *тому же самому* имени атрибута в двух и более путях. Если бы имя вызываемой функции в примере было другим в каждом пути, то вы могли бы применить операторы `from` и избежать повторения полного пути пакета при вызове одной из функций, как объяснялось ранее; расширение `as` в `from` тоже можно использовать для предоставления уникальных синонимов.

Кроме того, в показанной выше иерархии каталогов установки обратите внимание на то, что файлы `__init__.py` были добавлены в каталоги `system1` и `system2`, чтобы все заработало, но не в каталог `root`. Эти файлы требуются только в каталогах, перечисленных внутри операторов `import` в коде; как уже упоминалось, они выполняются автоматически, когда Python обрабатывает операции импортирования, проходя каталог пакета в первый раз.

Формально в данном случае каталог `system3` не обязан находиться под `root`, где должны размещаться только пакеты кода, из которых будет производиться импортование. Однако поскольку заранее неизвестно, когда ваши модули окажутся полезными в других программах, вы также можете поместить их под общий каталог `root` и избежать похожих проблем конфликта имен в будущем.

Наконец, имейте в виду, что операции импортирования двух исходных систем продолжат работать без изменений. Так как поиск производится сначала в их *домашних*

каталогах, добавление общего корневого каталога к пути поиска несущественно для кода в `system1` и `system2`. В нем можно применять просто `import utilities` и ожидать нахождения собственных файлов во время выполнения программ – хотя не при использовании в качестве пакетов в Python 3.X, как объясняется в следующем разделе. Если вы аккуратно распакуете все свои системы под общим корнем подобного рода, то конфигурация пути поиска тоже становится простой: вам понадобится только один раз добавить общий корневой каталог.

Что потребует внимания: пакеты модулей

Поскольку пакеты являются стандартной частью Python, обычно более крупные сторонние расширения поставляются как наборы каталогов пакетов, а не плоские списки модулей. Скажем, пакет Windows-расширений `win32all` для Python был одним из первых шагов в сторону повального увлечения пакетами. Многие его служебные модули находятся в пакетах, импортируемых с помощью путей. Например, для загрузки инструментов COM клиентской стороны применяется оператор следующего вида:

```
from win32com.client import constants, Dispatch
```

Он извлекает имена из модуля `client` пакета `win32com` – подкаталога установки.

Импортирование пакетов также широко распространено в коде, запускаемом под управлением Jython (основанной на Java реализации Python), потому что библиотеки Java организованы в иерархии. В недавних выпусках Python инструменты для работы с электронной почтой и XML аналогичным образом организованы в подкаталоги пакетов внутри стандартной библиотеки, а в Python 3.X в пакеты группируются даже более связанные модули – включая инструменты для построения графических пользовательских интерфейсов `tkinter`, инструменты для работы с сетями HTTP и т.д. Приведенные далее операции импортирования обеспечивают доступ к разнообразным стандартным библиотечным инструментам в Python 3.X (в Python 2.X использование может варьироваться):

```
from email.message import Message
from tkinter.filedialog import askopenfilename
from http.server import CGIHTTPRequestHandler
```

Создаете вы каталоги пакетов или нет, со временем вы наверняка будете их импортировать.

Относительное импортирование пакетов

До сих пор при раскрытии операций импортирования пакетов внимание было сосредоточено главным образом на импортировании файлов пакета *извне* пакета. Внутри самого пакета операции импортирования файлов этого же пакета могут применять тот же самый синтаксис полных путей, что у операций импортирования *извне* пакета – и как будет показано позже, иногда должны. Тем не менее, файлы пакета могут также задействовать специальные внутрипакетные правила поиска для упрощения операторов `import`. То есть вместо того, чтобы указывать список путей импортирования, операции импортирования внутри пакета могут быть относительными к этому пакету.

Работа такого приема зависит от версии: при выполнении операций импортирования Python 2.X неявно ищет сначала в каталогах пакетов, тогда как Python 3.X для импортирования из каталога пакета требует явного синтаксиса относительного импортирования. Такое изменение в Python 3.X способно улучшить читабельность кода,

делая операции импортирования в том же самом пакете более очевидными, но также привносит несовместимость с Python 2.X и может нарушить работу ряда программ.

Если вы начали изучение языка с версии Python 3.X, то ваше внимание в данном разделе, скорее всего, будет сосредоточено на его новом синтаксисе и модели импортирования. Однако если в прошлом вы использовали другие пакеты Python, тогда возможно вас также заинтересует, в чем отличия модули Python 3.X. Давайте выясним.



Как было показано в этом разделе, применение операций относительного импортирования пакетов фактически способно *ограничить роли ваших файлов*. Выражаясь кратко, они больше не могут использоваться как исполняемые файлы программ в Python 2.X и 3.X. По указанной причине во многих случаях нормальные пути импортирования пакетов могут оказаться наилучшим вариантом. Тем не менее, такая возможность сумела проникнуть во многие файлы Python и заслуживает оценки большинством программистов на Python, чтобы лучше понимать ее компромиссы и мотивацию.

Изменения в Python 3.X

Особенности работы операций импортирования внутри пакетов в Python 3.X слегка изменились. Изменение касается только операций импортирования внутри файлов, когда файлы применяются как часть каталога пакета; операции импортирования в других режимах использования работают прежним образом. Однако для *операций импортирования в пакетах* Python 3.X представляет два изменения.

- Он модифицирует семантику путей поиска модулей для импортирования, чтобы по умолчанию пропускать собственный каталог пакета. Операции импортирования проверяют только пути в `sys.path`. Они известны как операции абсолютного импортирования.
- Он расширяет синтаксис операторов `from`, предоставляя им возможность явно требовать, чтобы операторы импортирования проводили поиск только в каталоге пакета, с помощью ведущих точек. Это известно как синтаксис относительного импортирования.

Описанные изменения полностью присутствуют в линейке Python 3.X. Новый синтаксис оператора `from` для относительного импортирования также доступен в Python 2.X, но должен быть явно включен. Его включение может привести к нарушению работы программ Python 2.X, но доступно для прямой совместимости с Python 3.X.

Влияние этого изменения состоит в том, что в Python 3.X (и необязательно в Python 2.X) для импортирования модулей, находящихся в *том же* пакете, что и импортер, вы обычно должны применять специальный точечный синтаксис `from`. Исключением будут ситуации, когда в операциях импортирования указан полный путь относительно корня пакета в `sys.path` или операции импортирования относительны к домашнему каталогу файла верхнего уровня программы, где всегда производится поиск (которым, как правило, является текущий рабочий каталог).

Тем не менее, по умолчанию поиск в каталоге пакета автоматически не выполняется и внутрипакетные операции импортирования, предпринятые файлами в каталоге, который используется в качестве пакета, потерпят неудачу без специального синтаксиса `from`. Вы увидите, что в Python 3.X это может повлиять на то, как вы будете структурировать операции импортирования или каталоги для модулей, предназначенные для применения в программах верхнего уровня и в импортируемых пакетах. Однако для начала давайте посмотрим, как все это работает.

Основы относительного импортирования

В Python 3.X и 2.X операторы `from` теперь могут содержать ведущие точки (.) для указания на то, что им требуются модули, расположенные внутри того же самого пакета (что называется *операциями относительного импортирования*), а не модули, находящиеся где-то в другом месте в пути поиска импортируемых модулей (что называется *операциями абсолютного импортирования*).

- Операции импортирования с точками. В Python 3.X и 2.X вы можете использовать ведущие точки в именах модулей внутри операторов `from` для указания на то, что операции импортирования должны быть только относительными к включающему пакету. Такие операции импортирования будут выполнять поиск модулей только внутри каталога пакета и не искать модули с теми же именами, находящиеся в других местах в рамках пути поиска импортируемых модулей (`sys.path`). Совокупный эффект в том, что модули пакета перезаписывают внешние модули.
- Операции импортирования без точек. В Python 2.X нормальные операции импортирования в коде пакета, не содержащие ведущих точек, в настоящее время по умолчанию поддерживают относительный и затем абсолютный порядок поиска, т.е. они ищут сначала в собственном каталоге пакета. Тем не менее, в Python 3.X нормальные операции импортирования внутри пакета по умолчанию являются только абсолютными – при отсутствии любого специального точечного синтаксиса операции импортирования пропускают сам включающий пакет и ищут в других местах согласно пути поиска `sys.path`.

Например, в Python 3.X и 2.X оператор вида:

```
from . import spam # Относительно этого пакета
```

указывает Python на необходимость импортирования модуля по имени `spam`, расположенного в том же самом каталоге пакета, что и файл, в котором находится данный оператор. Подобным же образом оператор:

```
from .spam import name
```

означает “импортировать переменную `name` из модуля по имени `spam`, который расположен в том же самом пакете, что и файл, содержащий этот оператор”.

Поведение оператора *без* ведущих точек зависит от применяемой версии Python. В Python 2.X такая операция импортирования по-прежнему по умолчанию будет использовать исходный *относительный и затем абсолютный* порядок поиска (т.е. искать сначала в каталоге пакета), если только не поместить в начало импортирующего файла оператор следующего вида (как его первый исполняемый оператор):

```
from __future__ import absolute_import # Использовать в Python 2.X модель  
# относительного импортирования Python 3.X
```

Такой оператор активизирует изменение, внесенное в Python 3.X, для обеспечения *только абсолютного* пути поиска. В Python 3.X и в случае включения в Python 2.X операция импортирования без ведущей точки в имени модуля всегда приводит к тому, что Python пропускает относительные компоненты внутри пути поиска импортируемых модулей и взамен ищет в абсолютных каталогах, которые содержатся в `sys.path`. Скажем, в модели Python 3.X показанный ниже оператор всегда будет искать модуль `string` где-то в `sys.path`, а не модуль с тем же именем в пакете:

```
import string # Пропустить версию модуля из этого пакета
```

С другой стороны, без оператора `from __future__` в Python 2.X, если в пакете имеется локальный модуль `string`, тогда он и будет импортирован. Чтобы добиться одинакового поведения в Python 3.X и в Python 2.X, когда активизировано изменение, касающееся операций абсолютного импортирования, для принудительного применения операции относительного импортирования потребуется выполнить следующий оператор:

```
from . import string # Искать только в этом пакете
```

В настоящее время такой оператор работает как в Python 2.X, так и в Python 3.X. Единственное отличие модели Python 3.X в том, что данный оператор *обязателен*, если нужно загрузить модуль, который расположен в том же каталоге пакета, где находится содержащий этот оператор файл, когда он используется как часть пакета (и не указаны полные пути пакетов).

Обратите внимание, что ведущие точки могут применяться для принудительного применения операций относительного импортирования только в операторе `from`, но не в операторе `import`. В Python 3.X оператор `import имя_модуля` всегда выполняет операцию абсолютного импортирования, пропуская каталог включающего пакета. В Python 2.X такая форма оператора по-прежнему осуществляет операции относительного импортирования, производя поиск сначала в каталоге пакета. Операторы `from` без ведущих точек ведут себя так же, как операторы `import` – с только абсолютным порядком в Python 3.X (пропуск каталога пакета) и с относительным и затем абсолютным порядком в Python 2.X (поиск сначала в каталоге пакета).

Возможны также и другие схемы относительных ссылок, основанные на точках. Внутри файла модуля, расположенного в каталоге пакета по имени `turpkg`, приведенные далее альтернативные формы операции импортирования работают так, как описано в комментариях:

```
from .string import name1, name2 # Импортирует имена name1 и name2
                                # из turpkg.string
from . import string             # Импортирует turpkg.string
from .. import string            # Импортирует string из каталога
                                # того же уровня, что и turpkg
```

Чтобы лучше понять показанные формы и объяснить все эти добавленные сложности, нам необходимо двинуться окольным путем и выяснить подоплеку такого изменения.

Для чего используются операции относительного импортирования?

Помимо того, что внутрипакетные операции импортирования становятся более явными, данное средство предназначено отчасти для предоставления сценариям возможности устранять неоднозначности, которые могут возникать, когда файл с одним и тем же именем встречается в нескольких местах внутри пути поиска модулей. Взгляните на следующий каталог пакета:

```
turpkg\
  __init__.py
  main.py
  string.py
```

Здесь определен пакет по имени `turpkg`, содержащий модули с именами `turpkg.main` и `turpkg.string`. Теперь предположим, что модуль `main` пытается импортировать мо-

дуль по имени `string`. В Python 2.X и предшествующих версиях сначала просматривается каталог `туркг` для выполнения операции *относительного импортирования*. Будет найден и импортирован находящийся в каталоге `туркг` файл `string.py` с присваиванием его имени `string` в пространстве имен модуля `туркг.main`.

Однако вполне возможно, что целью такой операции импортирования была загрузка модуля `string` из стандартной библиотеки Python. К сожалению, в указанных версиях Python не существует прямолинейного способа проигнорировать `туркг.string` и искать модуль `string` стандартной библиотеки в соответствии с путем поиска модулей. Более того, нам не удастся выйти из положения с помощью полных путей импортирования пакетов, поскольку мы не можем зависеть от любой дополнительной структуры каталогов пакетов кроме стандартной библиотеки, присутствующей на каждом компьютере.

Другими словами, простые операции импортирования в пакетах могут быть неоднозначными и подверженными ошибкам. В рамках пакета совершенно не ясно, на что ссылается оператор `import spam` — на модуль внутри пакета или на модуль вне пакета. Как следствие, локальный модуль или пакет способен скрыть другой, доступный через `sys.path`, умышленно или нет.

На практике пользователи Python могут избегать выбора имен стандартных библиотечных модулей для собственных модулей (если вы нуждаетесь в стандартном модуле `string`, тогда не назначайте новому модулю имя `string!`). Но это не поможет, если пакет случайно скрывает стандартный модуль; кроме того, в будущем в Python могут добавить новый стандартный библиотечный модуль, который имеет такое же имя, как у вашего модуля. Код, опирающийся на операции относительного импортирования, также сложнее понимать, потому что читатель может запутаться, какой модуль планировалось использовать. Гораздо лучше, если распознавание можно делать явно в коде.

Решение проблемы с операциями относительного импортирования в Python 3.X

Чтобы выйти из затруднительного положения, операции импортирования, выполняемые внутри пакетов, в Python 3.X были изменены, став только абсолютными (что также доступно как вариант в Python 2.X). В рамках такой модели оператор `import` следующего вида в файле примера `туркг/main.py` будет всегда искать модуль `string` за пределами пакета через поиск абсолютного импортирования в `sys.path`:

```
import string          # Импортирует string извне пакета
                      # (абсолютное импортирование)
```

Операция импортирования `from` без синтаксиса с ведущими точками также трактуется как абсолютная:

```
from string import name      # Импортирует name из string извне пакета
```

Тем не менее, если вы действительно хотите импортировать модуль из своего пакета, не указывая его полный путь от корня пакета, то операции относительного импортирования все еще возможны в случае применения в операторе `from` синтаксиса с точками:

```
from . import string        # Импортирует здесь туркг.string
                           # (относительное импортирование)
```

Показанная форма оператора `from` импортирует модуль `string` только относительно текущего пакета и является эквивалентной операцией относительного импортирования для абсолютной версии предыдущего оператора `import` (оба оператора

загружают модуль целиком). Когда используется такой специальный синтаксис относительного импортирования, поиск производится только в каталоге пакета.

Мы также можем копировать отдельные имена из модуля посредством синтаксиса относительного импортирования:

```
from .string import name1, name2 # Импортирует имена name1 и name2  
# из mypkg.string
```

Оператор снова ссылается на модуль `string` относительно текущего пакета. Если этот код находится в модуле `mypkg.main`, например, тогда он будет импортировать `name1` и `name2` из `mypkg.string`.

На самом деле точка `(.)` в операции относительного импортирования обозначает каталог пакета, *содержащий* файл, в котором находится оператор импортирования. Дополнительная ведущая точка выполняет относительное импортирование, начиная с *родительского* каталога текущего каталога пакета. Скажем, следующий оператор:

```
from .. import spam # Импортирует из каталога того же уровня,  
# что и mypkg
```

загрузит модуль `spam` из каталога пакета того же уровня, что и `mypkg`. Выражаясь обобщенно, в коде модуля `A.B.C` можно применять любую из перечисленных ниже форм:

```
from . import D # Импортирует A.B.D (. означает A.B)  
from .. import E # Импортирует A.E (.. означает A)  
from .D import X # Импортирует A.B.D.X (. означает A.B)  
from ..E import X # Импортирует A.E.X (.. означает A)
```

Операции относительного импортирования или абсолютные пути пакетов

В качестве альтернативы иногда в файле может быть явно указано имя собственного пакета в операторе абсолютного импортирования относительно какого-то каталога из `sys.path`. Например, в следующем операторе поиск `mypkg` будет вестись в абсолютном каталоге из `sys.path`:

```
from mypkg import string # Импортирует mypkg.string  
# (абсолютное импортирование)
```

Однако такая форма полагается на настройки конфигурации и порядка в пути поиска модулей, в то время как синтаксис относительного импортирования – нет. Фактически эта форма требует, чтобы каталог, непосредственно содержащий `mypkg`, был включен в путь поиска модулей. Вероятно, так и будет, если `mypkg` является корнем пакета (либо иначе пакет не удастся использовать снаружи!), но данный каталог может быть вложенным в намного более крупное дерево пакетов. Если `mypkg` – не корень пакета, то в операторах абсолютного импортирования должны перечисляться все каталоги ниже элемента с корнем пакета в `sys.path`, когда имена пакетов явно указаны примерно так:

```
from system.section.mypkg import string # Контейнер system только в sys.path
```

В более крупных или глубоких пакетах работы по написанию кода может быть значительно больше, чем применение точки:

```
from . import string # Синтаксис относительного импортирования
```

С помощью последней формы поиск во включающем пакете производится автоматически независимо от настроек пути поиска, порядка в пути поиска и вложения каталогов. С другой стороны, форма с полными абсолютными путями будет работать

независимо от того, каким образом файл используется — как часть программы или как пакет, что мы будем исследовать далее.

Границы действия операций относительного импортирования

Операции относительного импортирования на первый взгляд могут показаться несколько озадачивающими, но понять их будет легче, если вы запомните ряд моментов о них.

- Операции относительного импортирования применимы только к импортированию внутри пакетов. Имейте в виду, что изменение пути поиска модулей, связанное с данным средством, применяется только к операторам импортирования внутри файлов модулей, используемых как часть пакета — т.е. внутрипакетных операций импортирования. Нормальные операции импортирования в файлах, не используемых в качестве части пакета, по-прежнему работают в точности, как было описано ранее, автоматически выполняя поиск сначала в каталоге, который содержит сценарий верхнего уровня.
- Операции относительного импортирования применимы только к оператору `from`. Также помните о том, что новый синтаксис этого средства применим только к операторам `from`, но не `import`. Он определяется тем фактом, что имя модуля в `from` начинается с одной и более точек. Имена модулей, которые содержат внутренние точки, но не имеют ведущей точки, обозначают операции импортирования пакетов, а не операции относительного импортирования.

Другими словами, операции относительного импортирования пакетов в Python 3.X на самом деле сводятся просто к устраниению поведения включающего пути поиска для пакетов, принятого в Python 2.X, наряду с добавлением специального синтаксиса `from`, который позволяет явно запрашивать поведение относительного импортирования только в пакетах. Если в прошлом вы писали операции импортирования пакетов таким образом, чтобы делать их независимыми от неявного относительного поиска в Python 2.X (например, всегда указывая полные пути от корня пакета), тогда это изменение будет в значительной степени спорным вопросом. Если вы поступали иначе, то вам придется обновить свои файлы пакетов с целью использования нового синтаксиса `from` для локальных файлов пакетов или полных абсолютных путей.

Сводка по правилам поиска модулей

С учетом пакетов и операций относительного импортирования история поиска модулей в Python 3.X, которую мы видели до сих пор, может быть подытожена следующим образом.

- Базовые модули с простыми именами (скажем, `A`) находятся за счет поиска в каждом каталоге из списка `sys.path` слева направо. Этот список создается из стандартных настроек системы и настроек, конфигурируемых пользователем, которые были описаны в главе 22.
- Пакеты представляют собой просто каталоги модулей Python со специальным файлом `__init__.py`, который делает возможным применение в операциях импортирования синтаксиса с путями к каталогам `A.B.C`. Например, в операции импортирования `A.B.C` каталог по имени `A` ищется относительно каталогов в `sys.path`, `B` является еще одним подкаталогом внутри `A` и `C` представляет собой модуль или другой импортируемый элемент в `B`.

- Внутри файлов пакета нормальные операторы `import` и `from` используют то же самое правило поиска в `sys.path`, что и операции импортирования в других местах. Тем не менее, операции импортирования в пакетах, реализованные с применением операторов `from` и ведущих точек, являются относительными к пакету, т.е. проверяется только каталог пакета, а обычный поиск в `sys.path` не используется. Скажем, в операторе `from . import A` поиск модуля ограничивается каталогом, содержащим файл, в котором находится данный оператор.

Python 2.X работает таким же образом за исключением того, что нормальные операции импортирования без точек также инициируют автоматический поиск сначала в каталоге пакета, после чего продолжают поиск в `sys.path`.

В целом операции импортирования Python выбирают между *относительным* поиском (во включающем каталоге) и *абсолютным* (в каталоге из `sys.path`) так, как описано ниже.

Операции импортирования с точками: `from . import m`, `from .m import x`

Только относительный поиск в Python 2.X и 3.X.

Операции импортирования без точек: `import m`, `from m import x`

Сначала относительный, затем абсолютный поиск в Python 2.X и только абсолютный в Python 3.X.

Как будет показано позже, в версии Python 3.3 у модулей появилась еще одна особенность — *пакеты пространств имен*, которые почти совершенно не связаны с раскрыываемой здесь историей о пакетах. Эта более новая модель тоже поддерживает операции импортирования относительно пакетов и является всего лишь другим способом создания пакета. Она дополняет процедуру поиска при импортировании, чтобы позволить содержимому пакета распространяться по множеству простых каталогов как последнее средство. Однако с точки зрения правил поиска в операциях относительного импортирования составной пакет впоследствии ведет себя аналогично.

Операции относительного импортирования в действии

Но достаточно теории: давайте выполним какой-нибудь простой код, чтобы продемонстрировать концепции, лежащие в основе операций относительного импортирования.

Операции импортирования за пределами пакетов

Прежде всего, как упоминалось ранее, данное средство не воздействует на операции импортирования за пределами пакета. Таким образом, следующий оператор найдет стандартный библиотечный модуль `string`, как и ожидалось:

```
C:\code> c:\Python37\python
>>> import string
>>> string
<module 'string' from 'C:\\\\Python37\\\\lib\\\\string.py'>
```

Но если мы добавим в каталог, где производится работа, модуль с таким же именем, то взамен будет выбран он, потому что первым элементом в пути поиска модулей является текущий рабочий каталог:

```
# code\\string.py
print('string' * 8)
```

```
C:\code> c:\Python37\python
>>> import string
stringstringstringstringstringstringstring
>>> string
<module 'string' from '.\\string.py'>
```

Другими словами, нормальные операции импортирования по-прежнему относительны к “домашнему” каталогу (контейнеру сценария верхнего уровня или каталогу, где вы работаете). На самом деле синтаксис относительного импортирования пакетов даже не разрешен в коде, который не находится в файле, являющемся частью пакета:

```
>>> from . import string
SystemError: Parent module '' not loaded, cannot perform relative import
Системная ошибка: родительский модуль '' не загружен, относительное
импортирование невозможно
```

В этом разделе код, введенный в интерактивной подсказке, ведет себя так же, как он вел бы себя в случае запуска из сценария верхнего уровня, поскольку первым элементом в `sys.path` будет либо рабочий каталог интерактивной подсказки, либо каталог, содержащий файл верхнего уровня. Единственное отличие в том, что `sys.path` начинается с абсолютного каталога, а не с пустой строки:

```
# code\main.py
import string                                # Тот же код, но в файле
print(string)

C:\code> C:\python37\python main.py    # Результаты в Python 2.X эквивалентны
stringstringstringstringstringstringstring
<module 'string' from 'c:\\code\\string.py'>
```

Подобным образом оператор `from . import string` в данном непакетном файле потерпит неудачу, как было в интерактивной подсказке – программы и пакеты представляют собой разные режимы использования файла.

Операции импортирования внутри пакетов

А теперь избавимся от локального модуля `string`, находящегося в текущем рабочем каталоге, и создадим там каталог пакета с двумя модулями, включающий обязательный, но пустой файл `code\pkg__init__.py`. Корни пакетов в настоящем разделе располагаются в текущем рабочем каталоге, автоматически добавляемом к `sys.path`, так что нам не придется устанавливать `PYTHONPATH`. Кроме того, ради экономии пространства пустые файлы `__init__.py` и большинство сообщений об ошибках приводятся не будут (а читателям с компьютерами, где установлена операционная система, отличающаяся от Windows, придется перенести показанные здесь команды оболочки на свои платформы):

```
C:\code> del string*    # del __pycache__\string* для байт-кода в Python 3.2+
C:\code> mkdir pkg
C:\code> notepad pkg\__init__.py

# code\pkg\spam.py
import eggs           # <== Работает в Python 2.X, но не в Python 3.X!
print(eggs.X)

# code\pkg\eggs.py
X = 99999
import string
print(string)
```

В первом файле пакета предпринимается попытка импортировать второй файл с помощью нормального оператора `import`. Из-за того, что это считается относительным в Python 2.X, но абсолютным в Python 3.X, во втором случае возникает ошибка. То есть Python 2.X ищет сначала в содержащем пакете, а Python 3.X не ищет. О таком *несовместимом поведении* следует помнить, имея дело с Python 3.X:

```
C:\code> c:\Python27\python
>>> import pkg.spam
<module 'string' from 'C:\Python27\lib\string.pyc'>
99999

C:\code> c:\Python37\python
>>> import pkg.spam
ModuleNotFoundError: No module named 'pkg'
Ошибка отсутствия модуля: модуль по имени pkg не найден
```

Чтобы заставить пример работать в *обеих* линейках Python 2.X и Python 3.X, модифицируем первый файл для применения специального синтаксиса относительного импортирования, так что в Python 3.X тоже будет выполняться поиск в каталоге пакета:

```
# code\pkg\spam.py
from . import eggs          # <== Использовать относительное импортирование
                             #   в Python 2.X или Python 3.X
print(eggs.X)

# code\pkg\eggs.py
X = 99999
import string
print(string)

C:\code> c:\Python27\python
>>> import pkg.spam
<module 'string' from 'C:\Python27\lib\string.pyc'>
99999

C:\code> c:\Python37\python
>>> import pkg.spam
<module 'string' from 'C:\\\\Python37\\\\lib\\\\string.py'>
99999
```

Операции импортирования по-прежнему относительны к текущему рабочему каталогу

В предыдущем примере обратите внимание на то, что модули пакета все еще имеют доступ к стандартным библиотечным модулям вроде `string` – их нормальные операции импортирования по-прежнему относительны к элементам в пути поиска модулей. В действительности, если вы снова добавите модуль `string` в текущий рабочий каталог, тогда операции импортирования в пакете найдут его там, а не в стандартной библиотеке. Хотя вы можете пропустить поиск в каталоге пакета посредством операции абсолютного импортирования в Python 3.X, вы не в состоянии пропустить поиск в домашнем каталоге программы, которая импортирует пакет:

```
# code\string.py
print("string" * 8)

# code\pkg\spam.py
from . import eggs
print(eggs.X)
```

```
# code\pkg\eggs.py
X = 99999
import string
print(string)
C:\code> c:\Python37\python    # Тот же самый результат, как и в Python 2.X
>>> import pkg.spam
stringstringstringstringstringstringstring
<module 'string' from '.\\string.py'>
99999
```

Выбор модулей с помощью операций относительного и абсолютного импортирования

Чтобы показать, как это применяется к операциям импортирования стандартных библиотечных модулей, снова переустановим пакет. Мы избавимся от локального модуля `string` и определим новый модуль внутри самого пакета:

```
C:\code> del string*    # del __pycache__\string* для байт-кода в Python 3.2+
# code\pkg\spam.py
import string      # <== Относительный в Python 2.X, абсолютный в Python 3.X
print(string)
# code\pkg\string.py
print('Ni' * 8)
```

Теперь то, какую версию модуля `string` вы получите, зависит от используемой версии Python. Как и ранее, Python 3.X считает операцию импортирования в первом файле абсолютной и пропускает поиск в пакете, но Python 2.X так не поступает – еще одно *несовместимое поведение* в Python 3.X:

```
C:\code> c:\Python37\python
>>> import pkg.spam
<module 'string' from 'C:\\Python37\\lib\\string.py'>
C:\code> c:\Python27\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\\string.py'>
```

Применение синтаксиса относительного импортирования в Python 3.X заставляет снова проводить поиск в пакете, как в Python 2.X – за счет использования синтаксиса абсолютного или относительного импортирования в Python 3.X вы можете явно либо пропускать, либо выбирать каталог пакета. На самом деле *это и является сценарием применения, который решает модель Python 3.X*:

```
# code\pkg\spam.py
from . import string    # <== Относительный в Python 2.X и 3.X
print(string)
# code\pkg\string.py
print('Ni' * 8)
C:\code> c:\Python37\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from '.\\pkg\\string.py'>
```

```
C:\code> c:\Python27\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Операции относительного импортирования выполняют поиск только в пакетах

Также важно отметить, что синтаксис относительного импортирования в действительности является *объявлением привязки*, а не просто предпочтением. Если теперь мы удалим файл `string.py` и любой ассоциированный байт-код в данном примере, то операция относительного импортирования в `spam.py` *померпит отказ* в Python 3.X и 2.X вместо того, чтобы использовать версию этого модуля из стандартной библиотеки (или любую другую):

```
# code\pkg\spam.py
from . import string      # <== Терпит неудачу в Python 2.X и 3.X,
                           #   если здесь отсутствует string.py!
C:\code> del pkg\string*
C:\code> C:\python37\python
>>> import pkg.spam
ModuleNotFoundError: No module named 'string'
Ошибка отсутствия модуля: модуль по имени string не найден

C:\code> C:\python27\python
>>> import pkg.spam
ImportError: cannot import name string
Ошибка импортирования: не удалось импортировать имя string
```

Модули, на которые производится ссылка в операциях относительного импортирования, должны существовать в каталоге пакета.

Операции относительного импортирования по-прежнему относительны к текущему рабочему каталогу (снова)

Несмотря на то что операции абсолютного импортирования позволяют таким способом пропускать модули пакета, они все еще полагаются на другие компоненты `sys.path`. В качестве последнего теста давайте определим два собственных модуля `string`. В следующем примере есть один модуль с таким именем в текущем рабочем каталоге, один в пакете и еще один в стандартной библиотеке:

```
# code\string.py
print('string' * 8)
# code\pkg\spam.py
from . import string      # <== Относительный в Python 2.X и 3.X
print(string)
# code\pkg\string.py
print('Ni' * 8)
```

Когда мы импортируем модуль `string` с помощью синтаксиса относительного импортирования вроде показанного, то ожидаемым образом получаем версию в пакете в обеих линейках Python 2.X и 3.X:

```
C:\code> c:\Python37\python    # В Python 2.X такой же результат
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from '.\\pkg\\string.py'>
```

Тем не менее, в случае применения синтаксиса абсолютного импортирования получаемый модуль снова варьируется в зависимости от версии. Python 2.X интерпретирует следующую операцию импортирования сначала как относительную к пакету, но Python 3.X делает ее “абсолютной”, что в данной ситуации просто означает пропуск ею поиска в пакете и загрузку версии, относительной к текущему рабочему каталогу – *не* версии из стандартной библиотеки:

```
# code\string.py
print('string' * 8)

# code\pkg\spam.py
import string          # <== Относительный в Python 2.X, "абсолютный"
                        #   в Python 3.X: текущий рабочий каталог!
print(string)

# code\pkg\string.py
print('Ni' * 8)

C:\code> c:\Python37\python
>>> import pkg.spam
stringstringstringstringstringstringstring
<module 'string' from '.\\string.py'>

C:\code> c:\Python27\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Как видите, хотя пакеты с помощью точек могут явно запрашивать модули внутри собственных каталогов, в противном случае их операции “абсолютного” импортирования по-прежнему относительны к остальным каталогам из нормального пути поиска модулей. В такой ситуации файл в программе, использующей пакет, скрывает стандартный библиотечный модуль, который может понадобиться пакету. Изменение в Python 3.X просто дает возможность коду пакета выбирать файлы либо внутри, либо снаружи пакета (т.е. относительно или абсолютно). Однако поскольку поиск при импортировании может зависеть от объемлющего контекста, спрогнозировать который нелегко, операции абсолютного импортирования в Python 3.X не являются гарантией нахождения модуля в стандартной библиотеке.

Поэкспериментируйте с приведенными примерами самостоятельно, чтобы лучше понять суть. На практике все обычно не настолько непродуманно, как может показаться: в целом на стадии разработки вы в состоянии структурировать свои операции импортирования, пути поиска и имена модулей для работы в желаемом стиле. Тем не менее, вы должны иметь в виду, что операции импортирования в крупных системах могут зависеть от контекста применения, а протокол импортирования модулей представляет собой часть успешного проектного решения библиотеки.

Затруднения, связанные с операциями импортирования относительно пакетов: смешанное использование

Теперь, получив представление об операциях импортирования относительно пакетов, вы также должны помнить о том, что они не всегда могут оказываться наилучшим вариантом. Операции абсолютного импортирования пакетов с полным путем относительно какого-то каталога в `sys.path` все еще временами предпочтительнее неявного импортирования относительно пакетов в Python 2.X и явного синтаксиса с точками при импортировании относительно пакетов в обеих линейках Python 2.X и Python 3.X.

Проблема может показаться не вполне ясной, но она довольно скоро может стать важной после того, как вы начнете создавать собственные пакеты.

Вы уже видели, что синтаксис относительного импортирования и принимаемое по умолчанию правило поиска по абсолютному пути в Python 3.X делают внутрипакетные операции импортирования явными, а потому более легкими для выявления и сопровождения, и делают возможным явный выбор в ряде сценариев с конфликтами имен. Однако эта модель имеет два важных последствия, о которых вы должны знать.

- В Python 3.X и 2.X применение операторов импортирования относительно пакетов неявно привязывает файл к каталогу и роли пакета и препятствует его использованию другими способами.
- В Python 3.X новое изменение правила поиска по относительному пути означает, что файл больше не может служить в качестве сценария и модуля пакета настолько же легко, как было в Python 2.X.

Причины указанных ограничений неуловимы, но это потому, что одновременно справедливо и следующее.

- Python 3.X и 2.X не разрешают применять синтаксис относительного импортирования `from ..`, если только импортер не используется как часть пакета (т.е. импортируется из какого-то другого места).
- Python 3.X не выполняет поиск модуля в собственном каталоге пакета для операций импортирования, если только не применяется синтаксис относительного импортирования `from .` (или модуль не находится в текущем рабочем каталоге либо в домашнем каталоге главного сценария).

Использование операций относительного импортирования препятствует созданию каталогов, которые служат в качестве места размещения исполняемых программ и внешне импортируемых пакетов в Python 3.X и 2.X. Кроме того, в Python 3.X некоторые файлы больше не могут служить сценариями и модулями пакетов, как было бы в Python 2.X. Что касается операторов импортирования, то правила выглядят показанным далее образом (первое предназначено только для *пакетного* режима в обеих линейках Python, а второе – для *программного* режима только в Python 3.X):

```
from . import mod      # Не разрешен в непакетном режиме в Python 2.X and 3.X
import mod              # Не производит поиск в собственном каталоге файла
                      #   в пакетном режиме в Python 3.X
```

Совокупный эффект в том, что для применения файлов в любой из линеек Python 2.X или Python 3.X, может потребоваться выбрать один режим использования – *пакетный* (с операциями относительного импортирования) или *программный* (с простым импортированием), а также изолировать настоящие файлы модулей пакетов в подкаталоге отдельно от файлов сценариев верхнего уровня.

В качестве альтернативы вы можете попытаться вручную изменять `sys.path` (в целом ненадежная и подверженная ошибкам задача) или всегда применять полные пути пакетов в операциях абсолютного импортирования вместо либо синтаксиса импортирования относительно пакетов, либо простого импортирования и предполагать наличие корня пакета в пути поиска модулей:

```
from system.section.mypkg import mod  # Работает в программном и пакетном режимах
```

Из всех описанных схем последняя – импортирование с полными путями пакетов – может быть самой переносимой и функциональной, но чтобы понять почему, нужно обратиться к более конкретному коду.

Проблема

Например, в Python 2.X один и тот же каталог обычно используется и как программа, и как пакет в случае применения нормальных операторов импортирования без точек. Когда каталог используется как программа, то поиск модулей для операций импортирования производится в домашнем каталоге сценария, а когда как пакет, то поиск модулей для внутрипакетных операций импортирования полагается на правило Python 2.X относительного и затем абсолютного поиска. Тем не менее, в Python 3.X прием работает не полностью — в пакетном режиме простые операции импортирования больше не загружают модули из того же самого каталога, если только это не контейнер файла или текущий рабочий каталог (и потому присутствует в `sys.path`).

Ниже все демонстрируется в действии с минимальным объемом кода (ради краткости здесь также не показаны файлы `__init__.py` каталогов пакета, обязательные до версии Python 3.3, и для разнообразия применяется запускающий модуль Windows, появившийся в Python 3.3):

```
# code\pkg\main.py
import spam

# code\pkg\spam.py
import eggs
# <== Работает, если находится в "."
#   домашний каталог главного сценария

# code\pkg\eggs.py
print('Eggs' * 4)
# Но не загружает этот файл, когда используется
#   как пакет в Python 3.X!

c:\code> python pkg\main.py # Нормально в случае применения как программы
#   в Python 2.X и 3.X
EggsEggsEggsEggs
c:\code> python pkg\spam.py
EggsEggsEggsEggs

c:\code> py -2 # Нормально в случае использования как пакета
#   в Python 2.X:
#   относительный, затем абсолютный
# Python 2.X: при простых операциях импортирования
#   поиск выполняется сначала в каталоге пакета

>>> import pkg.spam
EggsEggsEggsEggs

C:\code> py -3 # Но в Python 3.X терпит неудачу с поиском
#   файла здесь: только абсолютный
# Python 3.X: при простых операциях
#   импортирования поиск выполняется только
#   в текущем рабочем каталоге и sys.path
ModuleNotFoundError: No module named 'eggs'
Ошибка отсутствия модуля: модуль по имени eggs не найден
```

Следующим шагом могло бы стать добавление необходимого синтаксиса *относительного импортирования* для использования Python 3.X, но тут он не поможет. В приведенном далее решении сохраняется единственный каталог для главного сценария верхнего уровня и модулей пакета, а также добавляются требуемые точки. Теперь это работает в Python 2.X и 3.X, когда каталог импортируется как пакет, но терпит неудачу, когда он применяется как каталог программы (включая попытки запустить модуль напрямую в качестве сценария):

```
# code\pkg\main.py
import spam
```

```

# code\pkg\spam.py
from . import eggs      # <== Не пакет, если здесь присутствует главный сценарий!
# code\pkg\eggs.py
print('Eggs' * 4)

c:\code> python      # Нормально как пакет, но не программа в Python 3.X и 2.X
>>> import pkg.spam
EggsEggsEggsEggs

c:\code> python pkg\main.py
SystemError: ... cannot perform relative import
Системная ошибка: не удается выполнить относительное импортирование
c:\code> python pkg\spam.py
SystemError: ... cannot perform relative import
Системная ошибка: не удается выполнить относительное импортирование

```

Исправление 1: подкаталоги пакета

В случае смешанного использования подобного рода одно из решений предусматривает изоляцию в *подкаталоге* всех файлов кроме главного сценария, применяемого только программой. Тогда внутрипакетные операции импортирования продолжают работать во всех версиях Python, вы можете использовать верхний каталог как автономную программу, а вложенный каталог по-прежнему служит пакетом для применения из других программ:

```

# code\pkg\main.py
import sub.spam          # <== Работает, если переместить модули
                           # в пакет ниже главного сценария

# code\pkg\sub\spam.py
from . import eggs         # Теперь относительное импортирование пакета
                           # работает: в подкаталоге

# code\pkg\sub\eggs.py
print('Eggs' * 4)

c:\code> python pkg\main.py # Из главного сценария: одинаковые результаты
                           # в Python 2.X и 3.X
EggsEggsEggsEggs

c:\code> python             # Из какого-то другого места: одинаковые
                           # результаты в Python 2.X и 3.X
>>> import pkg.sub.spam
EggsEggsEggsEggs

```

Потенциальный недостаток такой схемы в том, что отсутствует возможность запуска модулей пакета напрямую, чтобы протестировать их посредством встроенного кода самотестирования, хотя взамен можно написать код тестов отдельно и поместить его в родительский каталог:

```

c:\code> py -3 pkg\sub\spam.py    # Но индивидуальные модули нельзя
                                 # запускать для тестирования
SystemError: ... cannot perform relative import
Системная ошибка: не удается выполнить относительное импортирование

```

Исправление 2: абсолютное импортирование с полными путями

В качестве альтернативы в данном случае подошел бы также синтаксис *импортирования пакетов с полными путями* — он требует наличия в пути поиска каталога, расположенного выше корня пакета, хотя в реалистичном программном пакете, вероят-

но, это не будет считаться дополнительным требованием. Большинство пакетов Python либо потребуют настройки пути поиска, либо обеспечат ее автоматически с помощью инструментов установки (таких как `distutils`, которые могут сохранять код пакета в каталоге, присутствующем в стандартном пути поиска модулей, наподобие `site-packages`; за деталями обращайтесь в главу 22):

```
# code\pkg\main.py
import spam

# code\pkg\spam.py
import pkg.eggs          # <== Полные пути пакетов работают во всех
                           # случаях, Python 2.X+3.X

# code\pkg\eggs.py
print('Eggs' * 4)

c:\code> set PYTHONPATH=C:\code
c:\code> python pkg\main.py    # Из главного сценария: одинаковые результаты
                           # в Python 2.X и 3.X
EggsEggsEggsEggs

c:\code> python            # Из какого-то другого места: одинаковые
                           # результаты в Python 2.X и 3.X
>>> import pkg.spam
EggsEggsEggsEggs
```

В отличие от исправления добавлением подкаталога абсолютное импортирование с полными путями вроде показанного также позволяет автономно запускать модули с целью тестирования:

```
c:\code> python pkg\spam.py    # Индивидуальные модули также допускают
                           # запуск в Python 2.X и 3.X
EggsEggsEggsEggs
```

Пример: приложение для самотестирования модуля (предварительный обзор)

Чтобы подвести итог, вот еще один типичный пример проблемы и ее решение посредством полных путей. Здесь используется распространенная методика, которую мы расширим в следующей главе, но идея достаточно проста для ее предварительного представления (правда, позже у вас может возникнуть желание возвратиться к этому предварительному обзору и снова пересмотреть его).

Рассмотрим приведенные ниже два модуля в каталоге пакета, из которых второй содержит код *самотестирования*. Выражаясь кратко, атрибут `__name__` модуля представляет собой строку '`__main__`', когда запускается как сценарий верхнего уровня, но не когда импортируется, что позволяет его применять в качестве модуля *и* сценария:

```
# code\dualpkg\m1.py
def somefunc():
    print('m1.somefunc')

# code\dualpkg\m2.py
...импортировать здесь m1...    # Заменить реальным оператором импортирования

def somefunc():
    m1.somefunc()
    print('m2.somefunc')
if __name__ == '__main__':
    somefunc()                  # Код самотестирования или режима
                                # использования как сценария верхнего уровня
```

Во втором модуле на месте заполнителя `...импортировать здесь m1...` необходимо импортировать первый модуль. Замена этой строки оператором относительного импортирования работает, когда файл используется как пакет, но в непакетном режиме не разрешена ни в Python 2.X, ни в Python 3.X (в целях экономии пространства результаты и сообщения об ошибках здесь не показаны; полный листинг доступен в файле `dualpkg\results.txt` внутри пакета примеров):

```
# code\dualpkg\m2.py
from . import m1

c:\code> py -3
>>> import dualpkg.m2          # Нормально
C:\code> py -2
>>> import dualpkg.m2          # Нормально
c:\code> py -3 dualpkg\m2.py    # Терпит неудачу!
c:\code> py -2 dualpkg\m2.py    # Терпит неудачу!
```

И наоборот, простой оператор импортирования работает в непакетном режиме в Python 2.X и 3.X, но терпит неудачу в Python 3.X, потому что в данной версии такие операторы не выполняют поиск в каталоге пакета:

```
# code\dualpkg\m2.py
import m1

c:\code> py -3
>>> import dualpkg.m2          # Терпит неудачу!
c:\code> py -2
>>> import dualpkg.m2          # Нормально
c:\code> py -3 dualpkg\m2.py    # Нормально
c:\code> py -2 dualpkg\m2.py    # Нормально
```

Наконец, применение полных путей пакетов снова работает в обоих режимах использования и обеих линейках Python при условии, что корневой каталог пакета присутствует в пути поиска модулей (как это должно быть, чтобы модуль можно было задействовать где-то в другом месте):

```
# code\dualpkg\m2.py
import dualpkg.m1 as m1          # И set PYTHONPATH=c:\code

c:\code> py -3
>>> import dualpkg.m2          # Нормально
C:\code> py -2
>>> import dualpkg.m2          # Нормально
c:\code> py -3 dualpkg\m2.py    # Нормально
c:\code> py -2 dualpkg\m2.py    # Нормально
```

В итоге, если только вы не хотите или не можете изолировать свои модули в подкаталогах ниже каталога сценариев, то операции импортирования с полными путями вероятно предпочтительнее операций импортирования относительно пакетов. Несмотря на больший объем набора, они обрабатывают все случаи и работают одинаково в линейках Python 2.X и Python 3.X. Могут существовать дополнительные обходные пути, которые предусматривают решение добавочных задач (например, ручная установка `sys.path` в коде), но здесь они не рассматриваются из-за того, что являются менее ясными и полагаются на подверженную ошибкам семантику импортирования. Операции импортирования с полными путями опираются только на базовый механизм пакетов.

Естественно, степень влияния этого на ваши модули может варьироваться от пакета к пакету; операции абсолютного импортирования могут требовать изменения при реорганизации каталогов, а операции относительного импортирования могут стать недействительными в случае перемещения локального модуля.



Не забывайте следить за изменениями в Python, происходящими в данной области.

Пакеты пространств имен, введенные в Python 3.3

Теперь, когда вы узнали все о пакетах и операциях импортирования относительно пакетов, необходимо объяснить, что появился новый вариант, который трансформирует ряд рассмотренных выше идей. Во всяком случае, формально, с выходом Python 3.3 стали доступными четыре модели импортирования. Ниже они перечислены, начиная с исходной и заканчивая самой новой.

Базовые операции импортирования модулей:

`import` модуль, `from` модуль `import` атрибут

Первоначальная модель: операции импортирования файлов и их содержимого, относительные к пути поиска модулей `sys.path`.

Операции импортирования пакетов: `import` каталог1.каталог2.модуль, `from` каталог1.модуль `import` атрибут

Операции импортирования, которые задают расширения путей к каталогам, относительные к пути поиска модулей `sys.path`, где каждый пакет содержитется в одиночном каталоге и имеет инициализационный файл, доступные в Python 2.X и 3.X.

Операции импортирования относительно пакетов: `from . import` модуль (относительное импортирование), `import` модуль (абсолютное импортирование)

Модель, применяемая для внутрипакетного импортирования, со схемами относительного и абсолютного поиска в операциях с точками и без точек, которая доступна, но отличается в линейках Python 2.X и Python 3.X.

Пакеты пространств имен: `import` разделенный_каталог.модуль

Новая модель пакетов пространств имен, введенная в Python 3.3, которая позволяет пакетам охватывать множество каталогов и не требует инициализационных файлов.

Первые две модели являются независимыми, третья ужесточает порядок поиска и расширяет синтаксис для внутрипакетного импортирования, а четвертая переворачивает ряд основных понятий и требований предшествующей модели импортирования пакетов. На самом деле в Python 3.3 и последующих версиях имеются две разновидности пакетов:

- исходная модель, теперь известная как обычные пакеты;
- альтернативная модель, известная как пакеты пространств имен.

По духу это похоже на противопоставление “традиционной” модели классов и модели классов “нового стиля”, которые рассматриваются в следующей части книги, хотя новая модель – нечто большее, нежели просто дополнение старой. Исходная и новая модели пакетов не являются взаимно исключающими и могут совместно использоваться в одной и той же программе. Фактически новая модель пакетов пространств имен работает как что-то вроде *запасного варианта*, который вступает в игру, только если в пути поиска модулей отсутствуют нормальные модули и обычные пакеты с тем же самым именем.

Логическое обоснование пакетов пространств имен произрастает из целей *установки* пакетов, которые могут казаться неясными, если только вы не отвечаете за решение таких задач, и лучше всего описано в документе PEP данного средства. Выражаясь кратко, пакеты пространств имен устраниют возможность конфликта множества файлов `__init__.py` при объединении частей пакета, полностью устранив такие файлы. Кроме того, за счет предоставления стандартной поддержки для пакетов, которые могут быть разнесены на множество каталогов и размещаться во множестве элементов `sys.path`, пакеты пространств имен повышают гибкость установки и предлагают общий механизм, заменяющий многие несовместимые решения, которые появлялись в попытке достигнуть этой цели.

Среднестатистические пользователи могут счесть пакеты пространств имен удобным альтернативным расширением модели обычных пакетов – они не требуют инициализационных файлов и позволяют использовать любой каталог кода как импортируемый пакет. Давайте перейдем к деталям.

Семантика пакетов пространств имен

Пакет пространства имен фундаментально не отличается от обычного пакета; он является лишь другим способом создания пакетов. Более того, они по-прежнему относительны `sys.path` на верхнем уровне: крайний слева компонент пути к пакету пространства имен с точками обязан находиться в элементе нормального пути поиска.

Однако с точки зрения физической структуры они могут существенно отличаться. Обычные пакеты должны иметь файл `__init__.py`, запускаемый автоматически, и размещаться в одном каталоге, как было ранее. По контрасту с ними пакеты пространств имен нового стиля *не могут* содержать файлы `__init__.py` и способны охватывать множество каталогов, которые объединяются на стадии импортирования. В действительности *ни один* из каталогов, составляющих пакет пространства имен, не может иметь файл `__init__.py`, но содержимое внутри них трактуется как единый пакет.

Алгоритм импортирования

Чтобы по-настоящему понять пакеты пространств имен, мы должны посмотреть на внутреннюю работу операции импортирования, начиная с версии Python 3.3. Во время импортирования Python по-прежнему проходит по всем каталогам в пути поиска модулей (который определяется `sys.path` для крайних слева компонентов операций абсолютного импортирования и местоположением пакета для операций относительного импортирования и компонентов, вложенных в пути пакета), как было в Python 3.2 и предшествующих версиях. Тем не менее, начиная с Python 3.3, во время поиска импортируемого модуля или пакета по имени `spam` для каждого каталога `directory` в пути поиска модулей Python проверяет более широкое многообразие критериев в следующем порядке.

1. Если найден файл `directory\spam__init__.py`, тогда импортируется и возвращается обычный пакет.
2. Если найден файл `directory\spam.{py, рус или другое расширение модуля}`, тогда импортируется и возвращается простой модуль.
3. Если найден каталог `directory\spam`, то он регистрируется и просмотр продолжается со следующего каталога в пути поиска.
4. Если ничего из перечисленного выше не найдено, тогда просмотр продолжается со следующего каталога в пути поиска.

Если просмотр пути поиска завершился без возвращения модуля или пакета на шаге 1 или 2 и без регистрации, по крайней мере, одного каталога на шаге 3, тогда создается *пакет пространства имен*.

Создание пакета пространства имен происходит немедленно и не откладывается до того, как встретится внутренняя операция импортирования. Новый пакет пространства имен имеет атрибут `__path__`, установленный в итерируемый объект со строками путей к каталогам, которые были обнаружены и зарегистрированы на шаге 3 просмотра, но не имеет атрибута `__file__`.

Атрибут `__path__` затем применяется в дальнейших более глубоких обращениях для поиска во всех компонентах пакета – всякий раз, когда запрашиваются дополнительные вложенные элементы, производится поиск в каждой зарегистрированной записи внутри атрибута `__path__` пакета пространства имен во многом подобно единственному каталогу обычного пакета.

Говоря по-другому, атрибут `__path__` пакета пространства имен исполняет ту же самую роль для компонентов более низкого уровня, что и `sys.path` на верхнем уровне для крайнего слева компонента путей к импортируемым пакетам. Он становится “родительским путем” для доступа к элементам более низкого уровня с использованием описанной выше процедуры из четырех шагов.

Совокупный эффект в том, что пакет пространства имен является своего рода *виртуальным скелетом* каталогов, находящихся через возможно многие элементы в пути поиска модулей. Однако после того как пакет пространства имен создан, функциональные отличия между ним и обычным пакетом отсутствуют; он поддерживает все, что присуще обычным пакетам, включая синтаксис импортирования относительно пакета.

Влияние на обычные пакеты: необязательность `__init__.py`

Одно из последствий новой процедуры импортирования связано с тем, что начиная с версии Python 3.3, пакеты больше не требуют файлов `__init__.py` – если в пакете с единственным каталогом данный файл отсутствует, тогда пакет будет трактоваться как пакет пространства имен с единственным каталогом без выдачи предупреждений. Это серьезное ослабление предшествующих правил, но обычно запрашиваемое изменение; многие пакеты не требуют инициализационного кода, и необходимость в создании пустых инициализационных файлов в таких случаях представляется излишней. С выпуском Python 3.3 файлы `__init__.py` стали необязательными.

В то же время исходная модель обычных пакетов по-прежнему полноценно поддерживается и автоматически выполняет код в `__init__.py` в качестве *привязки инициализации*. Кроме того, когда известно, что пакет никогда не будет частью расщепленного пакета пространства имен, то его реализация в виде обычного пакета с файлом `__init__.py` обеспечивает *преимущество в плане производительности*. Создание и за-

грузка обычного пакета происходит немедленно, как только он найден в пути поиска. Что касается пакетов пространств имен, то прежде чем пакет будет создан, должны быть просмотрены все элементы в пути поиска. Выражаясь более формально, обычные пакеты останавливают алгоритм, описанный в предыдущем разделе, на шаге 1, а пакеты пространств имен – нет.

Согласно документу PEP для данного изменения убирать поддержку обычных пакетов не планируется – по крайней мере, на сегодняшний день. Тем не менее, в проектах с открытым кодом изменение возможно всегда (на самом деле, в предыдущем издании книги цитировались планы относительно строкового форматирования и операций относительного импортирования в Python 2.X, которые позже были отменены), так что следите за будущей разработкой. Однако с учетом преимущества, связанного с производительностью, и кода автоматической инициализации обычных пакетов маловероятно, что такие пакеты когда-либо исчезнут.

Пакеты пространств имен в действии

Чтобы посмотреть, как работают пакеты пространств имен, создадим следующие два модуля иложенную структуру каталогов – с двумя подкаталогами по имени `sub`, расположенными в разных родительских каталогах, `dir1` и `dir2`:

```
C:\code\ns\dir1\sub\mod1.py  
C:\code\ns\dir2\sub\mod2.py
```

Если мы добавим каталоги `dir1` и `dir2` в путь поиска модулей, то `sub` станет пакетом пространства имен, охватывающим оба каталога, с двумя файлами модулей, доступными под этим именем, несмотря на то, что они находятся в разных физических каталогах. Ниже приведено содержимое файлов и обязательная настройка пути в Windows: здесь нет никаких файлов `__init__.py` – на самом деле они *не могут существовать* в пакетах пространств имен, т.к. это их главное физическое отличие:

```
c:\code> mkdir ns\dir1\sub          # Два каталога с одинаковыми именами  
c:\code> mkdir ns\dir2\sub          # в разных каталогах  
c:\code> type ns\dir1\sub\mod1.py  # Аналогично за рамками Windows  
print(r'dir1\sub\mod1')  
c:\code> type ns\dir2\sub\mod2.py  # Файлы модулей в разных каталогах  
print(r'dir2\sub\mod2')  
c:\code> set PYTHONPATH=C:\code\ns\dir1;C:\code\ns\dir2
```

Теперь при импортировании напрямую в Python 3.3 и последующих версиях пакет пространства имен является *виртуальным скелетом* своих индивидуальных компонент-каталогов и делает возможным доступ к более вложенным частям через единое составное имя с помощью нормальных операций импортирования:

```
c:\code> C:\Python37\python  
>>> import sub  
>>> sub                      # Пакеты пространств имен: вложенные пути поиска  
<module 'sub' (namespace)>  
>>> sub.__path__  
_NamespacePath(['C:\\code\\ns\\dir1\\sub', 'C:\\code\\ns\\dir2\\sub'])  
>>> from sub import mod1  
dir1\sub\mod1  
>>> import sub.mod2          # Содержимое из двух разных каталогов  
dir2\sub\mod2
```

```
>>> mod1
<module 'sub.mod1' from 'C:\\code\\ns\\dir1\\sub\\mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>
```

Сказанное также справедливо, если мы импортируем через имя пакета пространства имен *непосредственно* – поскольку пакет пространства имен создается, когда впервые достигается, синхронизация расширений пути к делу не относится:

```
c:\\code> C:\\Python37\\python
>>> import sub.mod1
dir1\\sub\\mod1
>>> import sub.mod2      # Один пакет охватывает два каталога
dir2\\sub\\mod2
>>> sub.mod1
<module 'sub.mod1' from 'C:\\code\\ns\\dir1\\sub\\mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>
>>> sub
<module 'sub' (namespace)>
>>> sub.__path__
[NamespacePath(['C:\\code\\ns\\dir1\\sub', 'C:\\code\\ns\\dir2\\sub'])]
```

Интересно отметить, что *операции относительного импортирования* тоже работают с пакетами пространств имен – следующий оператор относительного импортирования ссылается на файл в пакете, хотя указанный файл находится в *другом каталоге*:

```
c:\\code> type ns\\dir1\\sub\\mod1.py
from . import mod2      # И from . import string по-прежнему терпит неудачу
print(r'dir1\\sub\\mod1')

c:\\code> C:\\Python37\\python
>>> import sub.mod1      # Относительное импортирование mod2 из другого каталога
dir2\\sub\\mod2
dir1\\sub\\mod1
>>> import sub.mod2      # Уже импортированный модуль не выполняется повторно
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>
```

Как видите, пакеты пространств имен похожи на обычные пакеты с единственным каталогом во всех отношениях кроме наличия расщепленного физического хранилища – вот почему пакеты пространств имен с *единственным каталогом* без файлов `__init__.py` в точности подобны обычным пакетам, но без инициализационной логики, подлежащей выполнению.

Вложение пакетов пространств имен

Пакеты пространств имен даже поддерживают произвольное *вложение* – после того как пакет пространства имен создан, он на своем уровне исполняет по существу ту же самую роль, что и `sys.path` на верхнем уровне, становясь “родительским путем” для более низких уровней. Продолжим пример из предыдущего раздела:

```
c:\\code> mkdir ns\\dir2\\sub\\lower    # Дальнейшие вложенные компоненты
c:\\code> type ns\\dir2\\sub\\lower\\mod3.py
print(r'dir2\\sub\\lower\\mod3')

c:\\code> C:\\Python37\\python
```

```

>>> import sub.lower.mod3      # Пакет пространства имен, вложенный
                                # в пакет пространства имен
dir2\sub\lower\mod3
c:\code> C:\Python37\python
>>> import sub                  # Тот же самый эффект при постепенном доступе
>>> import sub.mod2
dir2\sub\mod2
>>> import sub.lower.mod3
dir2\sub\lower\mod3
>>> sub.lower                  # Пакет пространства имен с единственным каталогом
<module 'sub.lower' (namespace)>
>>> sub.lower._path_
	NamespacePath(['C:\\code\\ns\\dir2\\sub\\lower'])

```

В приведенном выше взаимодействии `sub` представляет собой пакет пространства имен, расщепленный на два каталога, а `sub.lower` – пакет пространства имен с единственным каталогом, который вложен в внутрь порции `sub`, физически размещенной в `dir2`. Вдобавок `sub.lower` – также пакет пространства имен, эквивалентный обычному пакету без файла `__init__.py`.

Такое поведение вложения сохраняется независимо от того, является компонент более низкого уровня модулем, обычным пакетом или еще одним пакетом пространства имен – выступая в качестве новых путей поиска для операций импортирования, пакеты пространств имен позволяют свободно вкладывать внутрь себя все перечисленные компоненты:

```

c:\code> mkdir ns\dir1\sub\pkg
C:\code> type ns\dir1\sub\pkg\__init__.py
print(r'dir1\sub\pkg\__init__.py')
c:\code> C:\Python37\python
>>> import sub.mod2          # Вложенный модуль
dir2\sub\mod2
>>> import sub.pkg           # Вложенный обычный пакет
dir1\sub\pkg\__init__.py
>>> import sub.lower.mod3   # Вложенный пакет пространства имен
dir2\sub\lower\mod3
>>> sub                      # Модули, пакеты и пространства имен
<module 'sub' (namespace)>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>
>>> sub.pkg
<module 'sub.pkg' from 'C:\\code\\ns\\dir1\\sub\\pkg\\__init__.py'>
>>> sub.lower
<module 'sub.lower' (namespace)>
>>> sub.lower.mod3
<module 'sub.lower.mod3' from 'C:\\code\\ns\\dir2\\sub\\lower\\mod3.py'>

```

Чтобы лучше понять суть, исследуйте файлы и каталоги рассмотренного примера. Легко заметить, что пакеты пространств имен плавно интегрированы в предшествующие модели импортирования и расширяют их новой функциональностью.

Файлы по-прежнему имеют приоритет над каталогами

Как объяснялось ранее, одна из целей файлов `__init__.py` в обычных пакетах состоит в том, чтобы объявить каталог пакетом. Наличие файла `__init__.py` сообщает Python о том, что необходимо использовать каталог, а не переходить дальше в поисках

возможного файла с тем же именем позже в пути. Такое решение позволяет избежать непреднамеренного выбора подкаталига, не относящегося к коду, который случайно появляется в начале пути поиска вместо желаемого модуля с тем же самым именем.

Из-за того, что пакеты пространств имен не требуют специальных файлов подобного рода, казалось бы, они сводят на нет указанную меру предосторожности. Тем не менее, это не так — поскольку описанный ранее алгоритм пространств имен продолжает просмотр пути после того, как каталог пространства имени был найден, файлы далее по пути по-прежнему имеют приоритет над встреченными ранее каталогами без файлов `__init__.py`. Например, создадим следующие каталоги и модули:

```
c:\code> mkdir ns2
c:\code> mkdir ns3
c:\code> mkdir ns3\dir
c:\code> notepad ns3\dir\ns2.py
c:\code> type ns3\dir\ns2.py
print(r'ns3\dir\ns2.py')
```

Каталог `ns2` не может быть импортирован в Python 3.2 и предшествующих версиях — он не является обычным пакетом, т.к. в нем отсутствует инициализационный файл `__init__.py`. Однако в Python 3.3 и последующих версиях данный каталог можно импортировать — он представляет собой каталог пакета пространства имен в текущем рабочем каталоге, которым всегда будет первый элемент в пути поиска модулей `sys.path` безотносительно к настройкам `PYTHONPATH`:

```
c:\code> set PYTHONPATH=
c:\code> py -3.2
>>> import ns2
ImportError: No module named ns2
Ошибка импортирования: отсутствует модуль по имени ns2
c:\code> py -3.7
>>> import ns2
>>> ns2      # Пакет пространства имен с единственным каталогом
           # в текущем рабочем каталоге
<module 'ns2' (namespace)>
>>> ns2.__path__
[NamespacePath(['.\.\ns2'])]
```

Но посмотрите, что происходит, когда каталог, содержащий файл с тем же самым именем, как у каталога пространства имен, добавляется *позже* в пути поиска, через настройку переменной `PYTHONPATH` — взамен будет использоваться этот файл, поскольку Python продолжает поиск по элементам далее в пути после нахождения каталога пакета пространства имен. Поиск останавливается только при обнаружении модуля или обычного пакета либо после того, как путь был полностью просмотрен. Пакеты пространств имен возвращаются, только если в ходе поиска ничего другого не было найдено:

```
c:\code> set PYTHONPATH=C:\code\ns3\dir
c:\code> py -3.7
>>> import ns2      # Использует найденный позже файл модуля,
                   # не каталог с таким же именем!
ns3\dir\ns2.py!
>>> ns2
<module 'ns2' from 'C:\\code\\ns3\\dir\\ns2.py'>
>>> import sys
>>> sys.path[:2]    # Первый элемент '' означает текущий рабочий каталог
[], 'C:\\code\\ns3\\dir']
```

Фактически установка пути для включения модуля работает таким же образом, как в ранних версиях Python, даже если аналогично названный каталог пространства имен появляется ранее в пути. Пакеты пространств имен применяются в Python 3.3 и последующих версиях только в случаях, которые в более ранних версиях Python были бы ошибками:

```
c:\code> py -3.2
>>> import ns2
ns3\dir\ns2.py!
>>> ns2
<module 'ns2' from 'C:\code\ns3\dir\ns2.py'>
```

По той же причине ни одному из каталогов в пакете пространства имен не разрешено иметь файл `__init__.py`: как только алгоритм импортирования находит такой каталог, он немедленно возвращает обычный пакет и прекращает поиск по пути и нахождение пакета пространства имен. Выражаясь более формально, алгоритм импортирования выбирает пакет пространства имен только в *конце* просмотра пути поиска и останавливается на шаге 1 или 2, если раньше обнаруживает обычный пакет или файл модуля.

Совокупный эффект заключается в том, что файлы модулей и обычные пакеты где угодно в пути поиска модулей имеют приоритет над каталогами пакетов пространств имен. Скажем, в следующем взаимодействии пакет пространства имен `sub` существует как сцепление каталогов с такими же именами под каталогами `dir1` и `dir2` в пути:

```
c:\code> mkdir ns4\dir1\sub
c:\code> mkdir ns4\dir2\sub
c:\code> set PYTHONPATH=c:\code\ns4\dir1;c:\code\ns4\dir2
c:\code> py -3
>>> import sub
>>> sub
<module 'sub' (namespace)>
>>> sub.__path__
[NamespacePath(['c:\\code\\ns4\\dir1\\sub', 'c:\\code\\ns4\\dir2\\sub'])]
```

Тем не менее, во многом подобно файлу модуля *обычный пакет*, добавленный к крайнему справа элементу в пути, тоже имеет приоритет перед аналогично называемыми каталогами пространств имен. Просмотр пути поиска начинает регистрацию предположительного пакета пространства имен в `dir1`, как было ранее, но прекращает ее, когда в `dir2` обнаруживается обычный пакет:

```
c:\code> notepad ns4\dir2\sub\__init__.py
c:\code> py -3
>>> import sub    # Использует позже найденный обычный пакет,
                  # а не каталог с таким же именем!
>>> sub
<module 'sub' from 'c:\\code\\ns4\\dir2\\sub\\__init__.py'>
```

Хотя пакеты пространств имен представляют собой полезное расширение, они доступны только читателям, которые работают с Python 3.3 и последующими версиями, а потому детали ищите в руководствах по Python. В частности, изучите документ PEP по этому изменению, где приведено логическое обоснование, дополнительные подробности и более полные примеры.

Резюме

В главе была представлена модель импортирования пакетов Python – необязательный, однако удобный способ явного перечисления части пути каталогов, ведущих к вашим модулям. Операции импортирования пакетов по-прежнему являются относительными к какому-то каталогу в пути поиска импортируемых модулей, но ваш сценарий явно задает остаток пути к модулю.

Вы видели, что пакеты не только делают операции импортирования более значащими в крупных системах, но также упрощают настройку пути поиска импортируемых модулей, если все операции импортирования между каталогами являются относительными к какому-то общему корневому каталогу. Кроме того, они устраниют неоднозначности, когда имеется более одного модуля с тем же самым именем – добавление имени включающего каталога к оператору импортирования помогает провести между ними различие.

Мы также исследовали новую модель *относительного импортирования*, имеющую значение только в коде внутри пакетов – способ явного выбора модулей в том же самом пакете с использованием ведущих точек в операторе `from` вместо того, чтобы полагаться на более старое и подверженное ошибкам неявное правило поиска пакетов. Наконец, мы рассмотрели появившиеся в версии Python 3.3 *пакеты пространств имен*, которые позволяют логическому пакету охватывать множество физических каталогов в качестве запасного варианта при поиске во время импортирования и устранить обязательность инициализационных файлов, принятую в предыдущей модели.

В следующей главе мы займемся несколькими более сложными темами, связанными с модулями, такими как переменная режима использования `__name__` и операции импортирования по строкам с именами. Но прежде чем двигаться дальше, закрепите пройденный материал главы, ответив на контрольные вопросы.

Проверьте свои знания: контрольные вопросы

1. Для чего предназначен файл `__init__.py` в каталоге пакета модуля?
2. Как можно избежать повторения полного пути пакета при ссылке на содержимое пакета?
3. Какие каталоги требуют наличия файлов `__init__.py`?
4. Когда с пакетами обязательно использовать оператор `import`, а не `from`?
5. В чем отличие между `from mypkg import spam` и `from . import spam`?
6. Что такое пакет пространства имен?

Проверьте свои знания: ответы

1. Файл `__init__.py` предназначен для объявления и инициализации обычного пакета модуля; Python автоматически выполняет его код, когда впервые проходит через каталог в процессе импортирования. Присвоенные в нем переменные становятся атрибутами созданного в памяти объекта модуля, который соответствует этому каталогу. Он также был обязательным в версиях, предшествующих Python 3.3 – если в каталоге отсутствовал файл `__init__.py`, то импортировать каталог с помощью синтаксиса пакетов было невозможно.

2. Применяйте оператор `from` с пакетом для копирования имен из пакета напрямую либо используйте расширение `as` с оператором `import` для назначения пути более короткого псевдонима. В обоих случаях путь указывается только в одном месте – в операторе `from` или `import`.
3. В Python 3.2 и предшествующих версиях каждый каталог, перечисленный в выполняемом операторе `import` или `from`, обязан содержать файл `__init__.py`. В других каталогах, включая каталог, который содержит крайний слева компонент пути пакета, этот файл необязателен.
4. Вы должны применять с пакетами оператор `import` вместо `from` только в случаях, когда необходим доступ к имени, которое определено в нескольких путях. В операторе `import` путь делает ссылки уникальными, но `from` допускает только одну версию любого заданного имени (если только вы не используете расширение `as` для переименования).
5. В Python 3.X оператор `from mypkg import spam` представляет собой операцию *абсолютного импортирования* – поиск для `mypkg` пропускает каталог пакета и модуль ищется в абсолютном каталоге внутри `sys.path`. С другой стороны, оператор `from . import spam` является операцией *относительного импортирования* – `spam` ищется только относительно пакета, в котором содержится этот оператор. В Python 2.X операция абсолютного импортирования производит поиск сначала в каталоге пакета, после чего продолжает его в `sys.path`; операции относительного импортирования работают так, как описано.
6. *Пакет пространства имен* – это расширение модели импортирования, доступное в Python 3.3 и последующих версиях, и соответствует одному и большему количеству каталогов, которые не содержат файлы `__init__.py`. Когда Python находит их во время поиска при импортировании и не обнаруживает первым простой модуль или обычный пакет, то создает пакет пространства имен, который представляет собой виртуальное скелетоне всех найденных каталогов, имеющих запрошенное имя модуля. Дальнейшие вложенные компоненты ищутся во всех каталогах пакета пространства имен. Результат подобен обычному пакету, но содержимое может охватывать множество каталогов.

Расширенные возможности модулей

Настоящая глава завершает эту часть книги рассмотрением совокупности более сложных тем, связанных с модулями – скрытие данных, модуль `_future_`, переменная `_name_`, изменения `sys.path`, инструменты для построения листингов пространств имен, импортирование модулей по строкам с именами, транзитивные перезагрузки и т.д. Кроме того, будет представлен стандартный набор распространенных ловушек и множество упражнений, в которых задействовано все то, что обсуждалось в данной части.

В ходе дела мы создадим несколько более крупных и полезных инструментов, нежели те, с которыми мы имели дело до сих пор, где скомбинируем функции и модули. Подобно функциям модули эффективнее, когда их интерфейсы правильно определены, поэтому в главе также приводится краткий обзор концепций проектирования модулей, часть которых мы уже исследовали в предшествующих главах.

Несмотря на наличие слова “расширенные” в названии главы, здесь главным образом рассматривается своего рода “сборная солянка” дополнительных тем о модулях. Поскольку некоторые из обсуждаемых здесь тем широко употребляются (особенно трюк с `_name_`), обязательно ознакомьтесь с ними, прежде чем переходить к изучению классов в следующей части книги.

Концепции проектирования модулей

Как и функции, модули демонстрируют проектные компромиссы: вы должны подумать о том, какие функции в каких модулях размещать, о механизмах взаимодействия между моделями и т.д. Все они станут яснее, когда вы приступите к написанию более крупных систем на Python, но ниже приводится несколько общих идей, которые следует иметь в виду.

- В Python вы всегда находитесь внутри модуля. Выражаясь кратко, просто не существует способов написания кода, который не будет находиться в каком-то модуле. Как бегло упоминалось в главах 17 и 21, даже код, набираемый в интерактивной подсказке, на самом деле попадает во встроенный модуль по имени `_main_`; уникальные особенности интерактивной подсказки заключаются лишь в том, что код выполняется и отбрасывается немедленно, а результаты выражений выводятся автоматически.

- **Сводите к минимуму связность модулей: глобальные переменные.** Подобно функциям модули работают лучше, если они реализованы как закрытые ящики. В качестве эмпирического правила запомните, что модули должны быть насколько возможно независимыми от глобальных переменных, используемых внутри других модулей, за исключением импортируемых из них функций и классов. Единственное, чем модуль должен делиться с внешним миром – это инструменты, которыми он пользуется, и инструменты, которые он определяет.
- **Доводите до максимума сцепление модулей: единая цель.** Вы можете минимизировать связность модулей за счет доведения до максимума их сцепления; если все компоненты модуля разделяют общую цель, тогда меньше шансов зависеть от внешних имен.
- **Модули должны редко изменять переменные других модулей.** Мы проиллюстрировали это с помощью кода в главе 17, но стоит здесь повторить: использовать глобальные переменные, определенные в другом модуле, вполне нормально (в конце концов, именно так клиенты импортируют службы), но изменение глобальных переменных из другого модуля часто является признаком наличия проблемы в проектном решении. Разумеется, существуют исключения, но вы должны стараться передавать результаты через такие механизмы, как аргументы функций и возвращаемые значения, а не через межмодульные изменения. В противном случае значения глобальных переменных становятся зависимыми от порядка выполнения операторов присваивания в других файлах, а модули будет труднее понять и многократно применять.

На рис. 25.1 приведена упрощенная схема среды, в которой работают модули. Модули содержат переменные, функции, классы и другие модули (если они импортированы).



Рис. 25.1. Среда выполнения модулей. Модули импортируются, но также способны импортировать и использовать другие модули, которые могут быть написаны на Python или на другом языке наподобие C. В свою очередь модули содержат переменные, функции и классы для выполнения своей работы, а их функции и классы могут содержать собственные переменные и другие элементы. Однако на верхнем уровне программы представляют собой просто наборы модулей

Функции имеют собственные локальные переменные, как и классы – объекты, существующие внутри модулей, которые начнут рассматриваться в следующей главе. Как было показано в части IV, функции тоже допускают вложение, но конечном итоге все содержится в модулях на верхнем уровне.

Сокрытие данных в модулях

Как мы уже видели, модуль Python экспортирует все имена, которым производится присваивание на верхнем уровне его файла. Понятие объявления, какие имена должны быть видны за пределами модуля, а какие нет, не существует. Фактически отсутствует способ запретить клиенту изменять имена внутри модуля, если у него возникнет такое желание.

В языке Python сокрытие данных в модулях является соглашением, а не синтаксическим ограничением. Если вы хотите нарушить работу модуля, уничтожив его имена, то вполне можете поступить так, но к счастью мне не приходилось встречать программиста, для которого это было бы жизненно важной целью. Некоторые сторонники пуританства возражают против такого свободного отношения к сокрытию данных, заявляя о том, что подобное положение вещей означает отсутствие возможности реализовать инкапсуляцию в Python. Тем не менее, инкапсуляция в Python больше связана с организацией пакетов, чем с установлением ограничений. В следующей части мы расширим эту идею в отношении классов, которые также не имеют синтаксиса закрытости, но часто способны эмулировать его в коде.

Сведение к минимуму вреда от `from *: _X и __all__`

В качестве особого случая вы можете снабжать имена префиксом в виде одиночного подчеркивания (например, `_X`), чтобы предотвратить их копирование, когда клиент импортирует имена модуля с помощью оператора `from *`. В действительности намерение такого действия заключается в том, чтобы свести к минимуму засорение пространства имен; поскольку `from *` копирует все имена, импортер может получить больше, чем он ожидал (в том числе имена, которые перезапишут имена в импортере). Подчеркивания не являются объявлениями “закрытых” имен: вы по-прежнему можете видеть и изменять такие имена посредством других форм импортирования, таких как оператор `import`:

```
# unders.py
a, _b, c, _d = 1, 2, 3, 4
>>> from unders import *      # Загружает только имена без подчеркиваний
>>> a, c
(1, 3)
>>> _b
NameError: name '_b' is not defined
Ошибка в имени: имя _b не определено
>>> import unders          # Но другие импортеры получают все имена
>>> unders._b
2
```

По-другому достичь эффекта сокрытия, похожего на соглашение по именованию `_X`, можно за счет присваивания переменной `__all__` списка со строками имен переменных на верхнем уровне модуля. В случае применения такой возможности оператор `from *` будет копировать только имена, перечисленные в списке `__all__`.

На самом деле это противоположность соглашению `_X`: `_all` идентифицирует имена, подлежащие копированию, в то время как `_X` обозначает имена, которые *не* должны копироваться. Python сначала ищет список `_all` в модуле и копирует его имена вне зависимости от наличия в них подчеркиваний; если переменная `_all` не определена, тогда `from *` копирует все имена, не содержащие одиночный ведущий символ подчеркивания:

```
# alls.py
__all__ = ['a', '_c']                                # __all__ имеет приоритет над _X
a, b, _c, _d = 1, 2, 3, 4
>>> from alls import *                            # Загружает только имена, перечисленные в
__all__
>>> a, _c
(1, 3)
>>> b
NameError: name 'b' is not defined
Ошибка в имени: имя b не определено
>>> from alls import a, b, _c, _d      # Но другие импортеры получают все имена
>>> a, b, _c, _d
(1, 2, 3, 4)
>>> import alls
>>> alls.a, alls.b, alls._c, alls._d
(1, 2, 3, 4)
```

Подобно соглашению `_X` список `_all` имеет смысл только для формы оператора `from *` и не эквивалентен объявлению закрытости: другие операторы импортирования по-прежнему могут получать доступ ко всем именам, как продемонстрировали последние две проверки. Однако разработчики модулей могут использовать любую из двух методик для реализации модулей, которые должным образом себя ведут, когда применяются вместе с `from *`. Вспомните также обсуждение списков `_all` внутри файлов `__init__.py` пакетов в главе 24; там эти списки объявляют подмодули, которые подлежат автоматической загрузке для оператора `from *` с их контейнером.

Включение будущих языковых средств: future

Изменения языка, потенциально способные нарушить работу существующего кода, обычно в Python вводятся постепенно. Они часто появляются как необязательные расширения, которые по умолчанию отключены. Для включения таких расширений используется специальный оператор `import` следующего вида:

```
from __future__ import название_средства
```

В файле сценария этот оператор должен быть первым исполняемым оператором (возможно, находясь после строки документации или комментария), потому что он включает специальную компиляцию кода для каждого модуля по отдельности. Такой оператор можно вводить и в интерактивной подсказке, чтобы поэкспериментировать с предстоящими изменениями языка; средство будет доступным до конца интерактивного сеанса.

Скажем, ранее в книге было показано, как применять данный оператор в Python 2.X для активизации настоящего деления из Python 3.X в главе 5, вызовов `print` из Python 3.X в главе 11 и операций абсолютного импортирования пакетов из Python

3.X в главе 24. В предшествующих изданиях книги эта форма оператора использовалась для демонстрации генераторных функций, требующих ключевого слова, которое еще не было по умолчанию включено (на месте `название_средства` указывалось `generators`).

Все изменения такого рода могут нарушить работу существующего кода в Python 2.X, поэтому они внедрялись постепенно или предлагались как необязательные расширения, включаемые с помощью описанного выше специального оператора `import`. В то же самое время определенные средства доступны для того, чтобы сделать возможным написание кода, обладающего прямой совместимостью с более поздними выпусками, куда вы когда-нибудь перенесете код.

Для просмотра списка будущих языковых средств, которые можно импортировать, выполните вызов `dir` на модуле `_future_` после его импортирования или поищите соответствующие сведения в руководстве по библиотеке. Согласно его документации названия будущих средств никогда не удалятся, поэтому совершенно безопасно оставлять импортирование `_future_` даже в коде, запускаемом под управлением версии Python, где такие средства присутствуют как нормальные.

Смешанные режимы использования: `__name__` и `__main__`

Следующий трюк, связанный с модулем, позволяет вам импортировать файл как модуль и запускать его как автономную программу; он широко применяется в файлах Python. В действительности он настолько прост, что некоторые поначалу упускают его из виду: каждый модуль имеет встроенный атрибут по имени `__name__`, который Python автоматически создает и присваивает, как описано ниже.

- Если файл запускается как программа верхнего уровня, тогда во время старта атрибут `__name__` устанавливается в строку "`__main__`".
- Если взамен файл импортируется, то атрибут `__name__` устанавливается в имя модуля, как известно его клиентам.

Результатом будет то, что модуль может проверять собственный атрибут `__name__` для выяснения, запущен он или импортирован. Например, пусть создан следующий файл модуля по имени `runme.py`, экспортирующий единственную функцию `tester`:

```
def tester():
    print("It's Christmas in Heaven...")
if __name__ == '__main__':
    # Только тогда, когда запущен
    tester()                         # Не тогда, когда импортирован
```

В модуле определена функция для импортирования и использования клиентами обычным образом:

```
c:\code> python
>>> import runme
>>> runme.tester()
It's Christmas in Heaven...
```

Но модуль в самом конце также содержит код, который настроен на автоматический вызов функции `tester`, когда данный файл запускается как программа:

```
c:\code> python runme.py
It's Christmas in Heaven...
```

В сущности, переменная `__name__` модуля служит *флагом режима использования*, позволяющим его коду быть задействованным как импортируемая библиотека и как сценарий верхнего уровня. Несмотря на простоту, вы увидите, что такая привязка применяется в большинстве программных файлов Python, с которыми вам придется сталкиваться в реальном мире – для тестирования и для двойного использования.

Скажем, пожалуй, самый распространенный вид применения проверки атрибута `__name__` предназначен для кода *самотестирования*. Выражаясь кратко, вы можете упаковать код, который тестирует экспортные средства модуля, в сам модуль за счет его помещения внутрь проверки атрибута `__name__` в конце файла. Таким образом, файл можно использовать в клиентах, *импортируя* его, но также тестировать его логику, запуская файл в командной оболочке или посредством другой схемы запуска.

Размещение кода самотестирования в конце файла внутри проверки атрибута `__name__` является, вероятно, наиболее часто применяемым и простым протоколом модульного тестирования в Python. Это гораздо удобнее, чем повторно набирать все тесты в интерактивной подсказке. (В главе 36 будут обсуждаться другие распространенные варианты для тестирования кода Python – как вы увидите, стандартные библиотечные модули `unittest` и `doctest` предлагают много расширенных инструментов тестирования.)

Вдобавок трюк с атрибутом `__name__` также часто используется при создании файлов, которые могут применяться как утилиты командной строки и как библиотеки инструментов. Например, предположим, что вы пишете на Python сценарий для поиска файлов. Вы можете извлечь из своего кода больше пользы, если упакуете его в функции и добавите в файл проверку атрибута `__name__` для автоматического вызова этих функций при запуске файла как автономной программы. В результате код сценария становится многократно используемым в других программах.

Модульное тестирование с помощью `__name__`

На самом деле в книге уже приводился хороший пример ситуации, где проверка атрибута `__name__` оказывается удобной. При рассмотрении аргументов в главе 18 мы писали сценарий, который вычислял минимальное значение из множества переданных аргументов (файл `minmax.py` в разделе “Функция `min`”):

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))      # Тестовый код
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

Сценарий содержит в конце код самотестирования, поэтому его можно тестировать, не набирая при каждом запуске аргументы в командной оболочке. Тем не менее, проблема такой реализации заключается в том, что результаты самотестирования будут выводиться всякий раз, когда файл импортируется в другом файле с целью применения в качестве инструмента – не вполне дружественная к пользователю особенность! Чтобы улучшить положение дел, мы можем поместить код самотестирования внутрь проверки атрибута `__name__`, так что он будет выполняться только в случае

запуска файла как сценария верхнего уровня, а не когда он импортируется (новая версия файла модуля переименована в min-max2.py):

```
print('I am:', __name__)
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

if __name__ == '__main__':
    print(minmax(lessthan, 4, 2, 1, 5, 6, 3))      # Тестовый код
    print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

Для контроля мы также выводим здесь значение атрибута `__name__`. Python создает и присваивает эту переменную режима использования, когда начинает загрузку файла. В случае запуска файла как сценария верхнего уровня атрибут `__name__` устанавливается в `__main__`, а потому выполняется код самотестирования:

```
c:\code> python minmax2.py
I am: __main__
1
6
```

Однако если файл импортируется, то атрибут `__name__` не содержит `__main__`, так что функцию потребуется вызывать явно:

```
c:\code> python
>>> import minmax2
I am: minmax2
>>> minmax2.minmax(minmax2.lessthan, 's', 'p', 'a', 'a')
'a'
```

Опять-таки независимо от того, применяется ли это для тестирования, совокупный эффект в том, что мы используем код в *двух разных ролях* – как модуль библиотеки инструментов или как исполняемую программу.



Согласно обсуждению операций относительного импортирования пакетов в главе 24 методика, описанная в настоящем разделе, способна оказывать определенное влияние на операции импортирования, выполняемые файлами, которые также применяются как компоненты пакетов в Python 3.X, но она по-прежнему может быть задействована с операциями импортирования по абсолютным путям пакетов и другими приемами. За дополнительными сведениями обращайтесь в предыдущую главу.

Пример: код с двойным режимом

Рассмотрим более сложный пример модуля, который демонстрирует еще один способ использования трюка с атрибутом `__name__` из предыдущего раздела. В показанном далее модуле `formats.py` определяются служебные функции форматирования строк для импортеров, а также производится проверка, запущен ли он как сценарий верхнего уровня. Если файл был запущен как сценарий, тогда проверяется наличие аргументов в командной строке; в случае их передачи запускается указанный тест, а иначе

че – заготовленный. Список `sys.argv` в Python содержит аргументы командной строки – он является списком строк, которые отражают слова, набранные в командной строке, причем первым элементом всегда будет имя выполняемого сценария. Мы применяли аргументы командной строки в качестве переключателей в инструменте оценочных испытаний из главы 21, но здесь они используется как общий механизм ввода:

```
#!python
"""
Файл: formats.py (Python 2.X и 3.X)
Разнообразные специализированные функции для форматирования строк
с целью отображения.
Тестирование можно производить с помощью заготовленного теста
или аргументов командной строки.
Что сделать: добавить круглые скобки для отрицательных денежных сумм,
реализовать больше возможностей.
"""

def commas(N):
    """
    Форматирует положительное целое число N для отображения
    с запятыми, разделяющими группы цифр: "xxx,yyy,zzz".
    """
    digits = str(N)
    assert digits.isdigit()
    result = ''
    while digits:
        digits, last3 = digits[:-3], digits[-3:]
        result = (last3 + ',' + result) if result else last3
    return result

def money(N, numwidth=0, currency='$'):
    """
    Форматирует число N для отображения с запятыми, 2 десятичными цифрами,
    ведущим символом $ и знаком, а также необязательным дополнением:
    "$ -xxx,yyy.zz".
    numwidth=0 - отсутствие дополнения пробелами,
    currency=' ' - опустить символ $
    или символ не ASCII для других валют (например, фунт - u'\xA3' или u'\u00A3').
    """
    sign = '-' if N < 0 else ''
    N = abs(N)
    whole = commas(int(N))
    fract = ('%.2f' % N)[-2:]
    number = '%s%s.%s' % (sign, whole, fract)
    return '%s%*s' % (currency, numwidth, number)

if __name__ == '__main__':
    def selftest():
        tests = 0, 1    # не проходит: -1, 1.23
        tests += 12, 123, 1234, 12345, 123456, 1234567
        tests += 2 ** 32, 2 ** 100
        for test in tests:
            print(commas(test))
        print('')
        tests = 0, 1, -1, 1.23, 1., 1.2, 3.14159
        tests += 12.34, 12.344, 12.345, 12.346
        tests += 2 ** 32, (2 ** 32 + .2345)
        tests += 1.2345, 1.2, 0.2345
```

```

tests += -1.2345, -1.2, -0.2345
tests += -(2 ** 32), -(2**32 + .2345)
tests += (2 ** 100), -(2 ** 100)
for test in tests:
    print('%s [%s]' % (money(test, 17), test))
import sys
if len(sys.argv) == 1:
    selftest()
else:
    print(money(float(sys.argv[1]), int(sys.argv[2])))

```

Код в файле работает одинаково в Python 2.X и 3.X. Когда файл запускается напрямую, он тестирует сам себя, но за счет указания параметров в командной строке можно управлять поведением тестирования. Запустите сценарий `formats.py` напрямую без аргументов командной строки, чтобы просмотреть вывод его кода самотестирования – из-за большого объема ниже приведена только часть вывода:

```

c:\code> python formats.py
0
1
12
123
1,234
12,345
123,456
1,234,567
... и так далее...

```

Для тестирования со специфическими строками передавайте их в командной строке вместе с минимальной шириной поля; код `__main__` сценария передаст их функции `money`, которая в свою очередь запустит `commas`:

```

C:\code> python formats.py 999999999 0
$999,999,999.00
C:\code> python formats.py -999999999 0
$-999,999,999.00
C:\code> python formats.py 123456789012345 0
$123,456,789,012,345.00
C:\code> python formats.py -123456789012345 25
$ -123,456,789,012,345.00
C:\code> python formats.py 123.456 0
$123.46
C:\code> python formats.py -123.454 0
$-123.45

```

Как и ранее, поскольку код снабжен возможностью работы в двойном режиме, мы также можем импортировать его инструменты обычным образом, чтобы применять их как библиотечные компоненты в сценариях, модулях и интерактивной подсказке:

```

>>> from formats import money, commas
>>> money(123.456)
'$123.46'
>>> money(-9999999.99, 15)
'$ -9,999,999.99'
>>> x = 999999999999999999999999
>>> '%s (%s)' % (commas(x), x)
'99,999,999,999,999,999,999 (999999999999999999999999)'

```

Аргументы командной строки можно использовать способами, аналогичными продемонстрированному в примере приему, чтобы снабдить обычным средством ввода сценарии, которые упаковывают свой код в виде функций и классов для многократного применения импортерами. Более сложная обработка аргументов командной строки описана в приложении А второго тома, а также в документации по модулям getopt, optparse и argparse в руководстве по стандартной библиотеке Python. В отдельных сценариях вы можете также использовать встроенную функцию input, которая применялась в главах 3 и 10, чтобы запрашивать тестовые входные данные у пользователя оболочки вместо их извлечения из командной строки. Дополнительные сведения об используемом здесь операторе assert ищите в главе 34 (том 2).



В главе 7 обсуждался синтаксис `{, d}` метода форматирования, появившийся в версиях Python 2.7 и 3.1; это расширение форматирования разделяет запятыми группы цифр во многом подобно тому, как было реализовано в приведенном выше коде. Тем не менее, предложенный здесь модуль добавляет форматирование денежных значений, может изменяться и служит ручной альтернативой вставкам запятых в ранних версиях Python.

Символы валют: Unicode в действии

Функция `money` модуля `formats` по умолчанию работает с денежными значениями в долларах, но поддерживает другие символы валют, позволяя передавать отличающиеся от ASCII символы Unicode. Например, порядковый номер Unicode с шестнадцатеричным значением 00A3 представляет собой символ фунта, а с 00A5 – иены. Указывать их можно в разнообразных формах, как описано далее.

- Порядковый номер (целое число) декодированной кодовой точки Unicode в текстовой строке, обозначив ее как Unicode или шестнадцатеричную (в Python 3.X для совместимости с Python 2.X такие строковые литералы должны содержать в своем начале `u`).
- Низкоуровневая закодированная форма символа в *байтовой строке*, которая декодируется перед передачей, обозначив ее как шестнадцатеричную (в Python 2.X для совместимости с Python 3.X такие строковые литералы должны содержать в своем начале `b`).
- Фактический символ в тексте программы вместе с объявлением кодировки исходного кода.

Предварительный обзор Unicode приводился в главе 4, а дополнительные детали будут предложены в главе 37, но базовые требования в данном примере довольно просты и являются подходящим сценарием применения. Для тестирования других символов валют мы поместим в файл `formats_currency.py` представленный далее код, чтобы не набирать его вручную в интерактивной подсказке каждый раз, когда вносится изменение:

```
from __future__ import print_function    # Python 2.X
from formats import money
X = 54321.987

print(money(X), money(X, 0, ''))
print(money(X, currency=u'\u00A3'), money(X, currency=u'\u00A5'))
print(money(X, currency=b'\u00A3'.decode('latin-1')))
print(money(X, currency=u'\u20AC'), money(X, 0, b'\u00A4'.decode('iso-8859-15')))
print(money(X, currency=b'\u00A4'.decode('latin-1')))
```

Ниже показано, как выглядит вывод этого тестового файла в IDLE версии Python 3.7 и в других надлежащим образом сконфигурированных контекстах. В Python 2.X тестовый файл работает точно так же, поскольку он выводит и кодирует строки переносимым образом. Как упоминалось в главе 11, импортирование `_future_` включает в Python 2.X вызовы `print` из Python 3.X. Кроме того, в главе 4 объяснялось, что байтовые литералы `b'...'` из Python 3.X в Python 2.X рассматриваются как простые строки, а Unicode-литералы `u'...'` из Python 2.X в Python 3.X трактуются как нормальные строки, начиная с версии Python 3.3.

```
$54,321.99 54,321.99  
£54,321.99 ₽54,321.99  
€54,321.99 Ⓜ54,321.99  
¤54,321.99
```

Если это работает на вашем компьютере, то вполне вероятно вы можете пропустить следующие несколько абзацев. Однако в зависимости от имеющихся настроек интерфейса и системы приведение кода в работоспособное состояние может требовать добавочных шагов. На моей машине он ведет себя корректно, когда Python и средства отображения синхронизированы, но в обычном окне командной строки Windows знак евро и обобщенный символ валюты в последних двух строках выводятся с ошибками.

В частности, тестовый сценарий всегда выполняется и производит вывод в графическом пользовательском интерфейсе *IDLE* из обеих линеек Python 3.X и Python 2.X благодаря хорошей поддержке отображения символов Unicode на их визуальные представления. Он также работает, как заявлено, в Python 3.X на компьютере Windows, если *перенаправить* вывод в файл, который затем открыть в Блокноте, потому что Python 3.X кодирует содержимое в стандартном формате Windows, который воспринимается Блокнотом:

```
c:\code> formats_currency.py > temp  
c:\code> notepad temp
```

Тем не менее, это не будет работать в Python 2.X, поскольку по умолчанию Python пытается декодировать выведенный текст как ASCII. Чтобы отображать символы, отличающиеся от ASCII, непосредственно в окне командной строки Windows, на некоторых компьютерах может понадобиться изменить *кодовую страницу* Windows (используемую для визуализации символов), а также установить переменную среды `PYTHONIOENCODING`, относящуюся к Python (применимую в качестве кодировки текста в стандартных потоках данных, в том числе при переводе символов в байты при выводе), в распространенный формат Unicode вроде UTF-8:

```
c:\code> chcp 65001          # Консоль соответствует Python  
c:\code> set PYTHONIOENCODING=utf-8 # Python соответствует консоли  
c:\code> formats_currency.py > temp # Python 3.X и 2.X записывают текст UTF-8  
c:\code> type temp             # Консоль отображает его корректно  
c:\code> notepad temp          # Блокнот тоже распознает UTF-8
```

На ряде платформ и даже в некоторых дистрибутивах Windows, возможно, вам не придется выполнять описанные выше шаги. Мне пришлось, потому что кодовая страница в моей системе установлена в 437 (символы для США), но ваша кодовая страница может быть другой.

Довольно тонкая причина того, что данный тест вообще работает в Python 2.X, заключается в том, что Python 2.X разрешает смешивать нормальные строки и строки Unicode при условии наличия в нормальных строках только 7-битных символов ASCII.

Начиная с Python 3.3, Unicode-литерал и '...' из Python 2.X поддерживается для совместимости, но сделан таким же, как нормальные строки '...', которые всегда являются строками Unicode (удаление ведущей буквы u обеспечивает работоспособность теста и в версиях Python 3.0–3.2, но нарушает совместимость с Python 2.X):

```
c:\code> py -2
>>> print u'\xA5' + '1', '%s2' % u'\u00A3'      # Python 2.X: смесь строк Unicode
# и нормальных строк с символами ASCII
$1 £2
c:\code> py -3
>>> print(u'\xA5' + '1', '%s2' % u'\u00A3')    # Python 3.X: нормальные строки
# являются строками Unicode, и '' необязательно
$1 £2
>>> print('\xA5' + '1', '%s2' % '\u00A3')
$1 £2
```

Как уже упоминалось, в главе 37 предлагается гораздо больше сведений по теме Unicode, которую многие видят второстепенной, но которая может неожиданно обнаружиться даже в относительно простых контекстах наподобие рассмотренного! Суть в том, что если оставить в стороне эксплуатационные проблемы, то внимательно написанный сценарий часто способен поддерживать Unicode в Python 3.X и 2.X.

Строки документации: документация по модулям в работе

Наконец, из-за того, что главном файле примера модуля используются *строки документации*, представленные в главе 15, мы можем также применять функцию `help` или режимы графического пользователяского интерфейса либо браузера PyDoc для исследования его инструментов – модули почти всегда автоматически являются инструментами общего назначения. Ниже показаны результаты работы функции `help`; на рис. 25.2 приведено представление PyDoc для нашего файла.

```
>>> import formats
>>> help(formats)
Help on module formats:

NAME
    formats

DESCRIPTION
    Файл: formats.py (Python 2.X и 3.X)
    Разнообразные специализированные функции для форматирования
    строк с целью отображения.
    Тестирование можно производить с помощью заготовленного теста
    или аргументов командной строки.
    Что сделать: добавить круглые скобки для отрицательных денежных сумм,
    реализовать больше возможностей.

FUNCTIONS
    commas(N)
        Форматирует положительное целое число N для отображения
        с запятыми, разделяющими группы цифр: "xxx,yyy,zzz".

    money(N, numwidth=0, currency='$')
        Форматирует число N для отображения с запятыми, 2 десятичными цифрами,
        ведущим символом $ и знаком, а также необязательным дополнением:
        "$ -xxx,yyy.zz".
```

`numwidth=0` - отсутствие дополнения пробелами,
`currency=''` - опустить символ \$
или символ не ASCII для других валют
(например, фунт - `u'\xa3'` или `u'\u20ac'`).

FILE

c:\code\formats.py

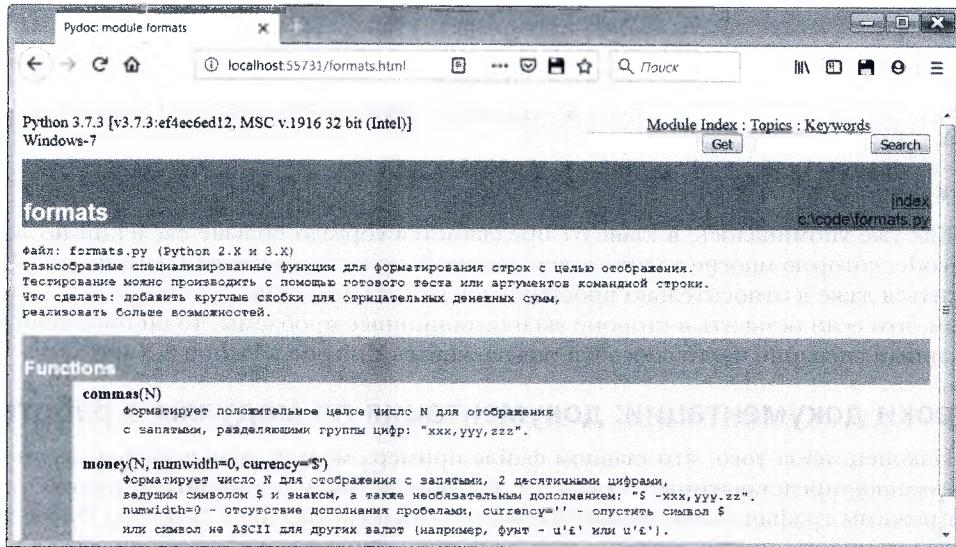


Рис. 25.2. Представление PyDoc для файла `formats.py`, полученное за счет выполнения командной строки `py -3 -m pydoc -b` в Python 3.2 и последующих версиях и щелчка на индексной записи для этого файла (см. главу 15)

Изменение пути поиска модулей

Давайте возвратимся к более общим темам о модулях. В главе 22 мы выяснили, что путь поиска модулей представляет собой список каталогов, который может настраиваться через переменную среды `PYTHONPATH` и возможно через файлы `.pth`. Что до сих пор не было показано — так это то, как на самом деле изменять список поиска в самой программе Python, модифицируя встроенный список `sys.path`. В соответствии с главой 22 список `sys.path` инициализируется во время начального запуска, но впоследствии вы можете удалять, добавлять и сбрасывать его компоненты желаемым образом:

```
>>> import sys
>>> sys.path
['', 'c:\\temp', 'C:\\Windows\\system32\\python37.zip', ...остальное удалено...]
>>> sys.path.append('C:\\sourcedir')      # Расширение пути поиска модулей
>>> import string                      # Все операции импортирования производят поиск
                                         # в новом каталоге в последнюю очередь
```

После того, как такое изменение внесено, оно будет воздействовать на все будущие операции импортирования во время выполнения программы Python, поскольку все импортеры разделяют тот же самый одиночный список `sys.path` (во время выпол-

нения программы в памяти присутствует только одна копия отдельно взятого модуля — вот для чего существует функция `reload`). Фактически список `sys.path` можно произвольно изменять:

```
>>> sys.path = [r'd:\temp']           # Изменение пути поиска модулей
>>> sys.path.append('c:\\lp5e\\examples') # Только для этого запуска (процесса)
>>> sys.path.insert(0, '... ')
>>> sys.path
['...', 'd:\\temp', 'c:\\lp5e\\examples']
>>> import string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'string'
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
Ошибка отсутствия модуля: модуль по имени string не найден
```

Таким образом, вы можете использовать эту методику для динамического конфигурирования пути поиска внутри программы Python. Однако будьте осторожны: если вы удалите критически важный каталог из пути поиска, то можете утратить доступ к нужным служебным средствам. Скажем, в предыдущем примере мы больше не имеем доступа к модулю `string`, потому что удалили из пути поиска каталог библиотеки Python!

Кроме того, не забывайте о том, что такие настройки `sys.path` сохраняются только на протяжении функционирования сеанса или программы Python (формально *процесса*), где они были сделаны; после выхода из Python они утрачиваются. Напротив, конфигурации пути поиска в переменной `PYTHONPATH` и файлах `.pth` находятся в среде операционной системы, а не в выполняющейся программе Python, и потому более глобальны: они подхватываются каждой программой на компьютере и существуют после ее завершения. В некоторых системах первый вариант может быть организован на основе пользователей, а второй охватывать всю установку.

Расширение `as` для операторов `import` и `from`

Со временем операторы `import` и `from` были расширены, чтобы позволить назначать импортированному имени другое имя в сценарии. Мы применяли это расширение раньше, но есть ряд дополнительных деталей. Следующий оператор `import`:

```
import modulename as name      # Использовать name, не modulename
```

эквивалентен приведенному ниже коду, который переименовывает модуль только в области видимости импортера (другим файлам модуль будет известен под своим первоначальным именем):

```
import modulename
name = modulename
del modulename      # Не сохранять первоначальное имя
```

После такого оператора `import` для ссылки на модуль вы можете — и фактически должны — использовать имя, указанное после `as`. Прием также работает в операторе `from`, позволяя назначать имени, которое импортировано из файла, другое имя в области видимости импортера; как и ранее, вы получите только указанное новое имя, но не первоначальное:

```
from modulename import attrname as name      # Использовать name, не attrname
```

Как обсуждалось в главе 23, это расширение часто применяется с целью предоставления *кратких псевдонимов* для более длинных имен и устранения *конфликтов имен*, когда в сценарии уже используется имя, которое иначе было бы перезаписано обычным оператором `import`:

```
import reallylongmodulename as name      # Использовать короткий псевдоним
name.func()

from module1 import utility as util1      # Можно иметь только одно имя utility
from module2 import utility as util2
util1(); util2()
```

Расширение `as` также оказывается полезным, когда нужно предоставить короткое простое имя для целого пути к каталогу и избежать конфликтов имен в случае применения средства импортирования пакетов, описанного в главе 24:

```
import dirl.dir2.mod as mod              # Полный путь указывается только один раз
mod.func()

from dirl.dir2.mod import func as modfunc # При необходимости переименовать,
                                            # чтобы сделать уникальным
modfunc()
```

Прием также является своего рода защитой от изменения имен: если в новом выпуске библиотеки модуль или инструмент, широко используемый в вашем коде, получает новое имя, либо предлагается альтернатива, которую необходимо применять взамен, тогда вы при импортировании всего лишь назначаете ему прежнее имя и предотвращаете нарушение работоспособности имеющегося кода:

```
import newname as oldname
from library import newname as oldname
...и продолжить использовать oldname до тех пор, пока не появится время
обновить весь код...
```

Скажем, такой подход может справиться с некоторыми изменениями в библиотеке Python 3.X (например, `tktinter` из Python 3.X против `Tkinter` из Python 2.X), хотя часто они представляют собой нечто большее, чем просто новые имена!

Пример: модули являются объектами

Поскольку модули делают видимым большинство своих интересных свойств в форме встроенных атрибутов, легко писать программы, которые управляют другими программами. Такие управляющие программы мы обычно называем *метапрограммами*, потому что они работают поверх других систем. На это также ссылаются как на *самоанализ*, т.к. программы способны видеть и обрабатывать внутренности объектов. Самоанализ – довольно сложное средство, но оно может быть полезным для построения программных инструментов.

Например, чтобы получить атрибут по имени `name` из модуля `M`, мы можем использовать уточнение с помощью атрибута или индекс в словаре атрибутов модуля, доступном через встроенный атрибут `__dict__`, который обсуждался в главе 23. Python также экспортирует список всех загруженных модулей в виде словаря `sys.modules` и предоставляет встроенную функцию `getattr`, которая позволяет извлекать атрибуты по строкам с их именами, что похоже на `object.attr`, но только `attr` является выра-

жением, которое выдает строку во время выполнения. Из-за этого все перечисленные далее выражения попадают на тот же самый атрибут и объект¹:

```
M.name          # Уточнение объекта атрибутом
M.__dict__['name']    # Ручная индексация словаря пространства имен
sys.modules['M'].name  # Ручная индексация таблицы загруженных модулей
getattr(M, 'name')     # Вызов встроенной функции извлечения
```

Делая видимыми внутренности модулей подобным образом, Python помогает строить программы о программах. Скажем, ниже показан код файла модуля по имени `mydir.py`, который воплощает описанные идеи в реализации настроенной версии встроенной функции `dir`. В нем определяется функция `listing`, которая принимает в качестве аргумента объект модуля и выводит форматированный листинг пространства имен этого модуля с сортировкой по имени:

```
#!/usr/bin/python
"""
mydir.py: модуль, который выводит листинг пространства имен другого модуля
"""

from __future__ import print_function      # Совместимость с Python 2.X
sepflen = 60
sepchr = '-'

def listing(module, verbose=True):
    sepline = sepchr * sepflen
    if verbose:
        print(sepline)
        print('name:', module.__name__, 'file:', module.__file__)
        print(sepline)
    count = 0
    for attr in sorted(module.__dict__):      # Просмотр ключей пространств
                                                # имен (или перечисление)
        print('%02d) %s' % (count, attr), end = ' ')
        if attr.startswith('__'):
            print('<built-in name>')           # Пропуск __file__ и т.д.
        else:
            print(getattr(module, attr))       # То же, что и __dict__[attr]
        count += 1
    if verbose:
        print(sepline)
        print(module.__name__, 'has %d names' % count)
        print(sepline)
if __name__ == '__main__':
    import mydir
    listing(mydir)                          # Код самотестирования: список для самого себя
```

¹ Как кратко упоминалось в разделе “Другие способы доступа к глобальным переменным” главы 17, поскольку функция может получать доступ к ее включающему модулю через таблицу `sys.modules` вроде этой, она также может использоваться для эмуляции эффекта от оператора `global`. Например, эффект от `global X; X=0` можно эмулировать (хотя и за счет набора намного большего объема кода!), поместив внутрь функции следующие операторы: `import sys; glob=sys.modules[__name__]; glob.X=0`. Не забывайте, что каждый модуль получает атрибут `__name__` бесплатно; он является видимым как глобальное имя внутри функций в рамках модуля. Данный трюк предоставляет еще один способ изменения локальных и глобальных переменных с теми же самыми именами внутри функций.

Обратите внимание на строку документации в начале. Как и в предыдущем примере formats.py, из-за того, что мы хотим применять его как универсальный инструмент, строка документации предоставляет функциональную информацию, доступную через help и режимы графического пользовательского интерфейса или браузера PyDoc – инструмента, который в своей работе использует похожие средства самоанализа. В конце модуля также предоставляется код *самотестирования*, который импортирует и строит листинг для самого модуля mydir. Ниже приведен вывод, полученный в Python 3.7; сценарий работает и в Python 2.X (где листинг может содержать меньшее количество имен) по причине импортирования функции вывода из __future__:

```
c:\code> py -3 mydir.py
-----
name: mydir file: C:\Code\mydir.py
-----
00) __builtins__ <built-in name>
01) __cached__ <built-in name>
02) __doc__ <built-in name>
03) __file__ <built-in name>
04) __loader__ <built-in name>
05) __name__ <built-in name>
06) __package__ <built-in name>
07) __spec__ <built-in name>
08) listing <function listing at 0x02DBAAE0>
09) print_function _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0),
65536)
10) sepchr -
11) seplen 60
-----
mydir has 12 names
```

Чтобы применять модуль mydir в качестве инструмента получения листингов для других модулей, функции listing этого модуля необходимо передавать объекты других модулей. Далее показан листинг атрибутов в модуле для построения графических пользовательских интерфейсов tkinter (Tkinter в Python 2.X) из стандартной библиотеки; формально mydir будет работать с любым объектом, имеющим атрибуты __name__, __file__ и __dict__:

```
>>> import mydir
>>> import tkinter
>>> mydir.listing(tkinter)
-----
name: tkinter file: F:\Python37\lib\tkinter\__init__.py
-----
00) ACTIVE active
01) ALL all
02) ANCHOR anchor
03) ARC arc
04) BASELINE baseline
...остальные имена не показаны...
155) image_types <function image_types at 0x00E784B0>
156) mainloop <function mainloop at 0x00E64108>
157) re <module 're' from 'F:\Python37\lib\re.py'>
158) sys <module 'sys' (built-in)>
159) wantobjects 1
```

```
-----  
tkinter has 160 names  
-----
```

Позже мы снова встретим функцию `getattr` и родственные инструменты. Здесь важно отметить, что `mydir` является программой, которая позволяет просматривать другие программы. Поскольку Python делает видимыми внутренности объектов, вы можете их обрабатывать обобщенным образом².

Импортирование модулей по строкам с именами

Имя модуля в операторе `import` или `from` является жестко закодированным именем переменной. Тем не менее, иногда ваша программа будет получать имя модуля, подлежащего импортированию, в виде строки во время выполнения — например, в результате выбора, производимого в графическом пользовательском интерфейсе, или разбора XML-документа. К сожалению, вы не можете использовать операторы `import` напрямую для загрузки модуля с именем, заданным как строка — Python ожидает имя переменной, которое берется буквально и не оценивается, а не строку или выражение. Вот пример:

```
>>> import 'string'  
File "<stdin>", line 1  
    import 'string'  
          ^  
  
SyntaxError: invalid syntax  
Синтаксическая ошибка: недопустимый синтаксис
```

Простое присваивание имени переменной соответствующей строки тоже не поможет:

```
x = 'string'  
import x
```

Здесь Python будет пытаться импортировать файл `x.py`, а не модуль `string` — имя, указанное в операторе `import`, становится переменной, которой присваивается объект загруженного модуля, и буквально идентифицирует внешний файл.

Выполнение строк с кодом

Чтобы обойти это, вам придется применять специальные инструменты для загрузки модуля динамически из строки, которая генерируется во время выполнения. Наиболее общий подход предусматривает создание оператора `import` в виде строки кода Python и ее передачи встроенной функции `exec` для выполнения (`exec` в Python 2.X является оператором, но его можно использовать в точности, как тут показано — круглые скобки попросту игнорируются):

² Вы можете предварительно загружать `mydir.listing` и рассматриваемый позже инструмент перезагрузки в пространство имен интерактивного сеанса, импортируя их в файле, на который имеется ссылка в переменной среды `PYTHONSTARTUP`. Из-за того, что код файла начального запуска выполняется в пространстве имен интерактивного сеанса (модуль `__main__`), импортирование общих инструментов в данном файле может уменьшить объем набора. Дополнительные сведения ищите в приложении А (том 2).

```
>>> modname = 'string'
>>> exec('import ' + modname)      # Выполнение строки кода
>>> string                      # Импортируется в это пространство имен
<module 'string' from 'F:\\Python37\\lib\\string.py'>
```

Мы встречали функцию `exec` (и родственный ей вариант для выражений, `eval`) ранее в главах 3 и 10. Она компилирует строку кода и передает ее на выполнение интерпретатору Python. В Python компилятор байт-кода доступен во время выполнения, так что вы можете писать программы, которые создают и запускают другие программы. По умолчанию функция `exec` выполняет код в текущей области видимости, но при необходимости вы можете указать более конкретную информацию, передавая ей необязательные слова `exec` пространств имен. Она также порождает упомянутую ранее в книге проблему в плане безопасности, которая может быть незначительной в строке с кодом, создаваемой вами самостоятельно.

Прямые вызовы: два варианта

Единственный реальный недостаток применения функции `exec` в данной ситуации связан с тем, что при каждом выполнении она должна компилировать оператор `import`, что может быть медленным. Предварительная компиляция в байт-код посредством встроенной функции `compile` может помочь выполнять строки с кодом много раз, но в большинстве случаев вероятно проще использовать встроенную функцию `__import__`, чтобы загрузить модуль, имя которого указано в строке; вдобавок она может выполняться быстрее (см. главу 22). Эффект будет похожим, но функция `__import__` возвращает объект модуля, поэтому для сохранения потребуется присваивание:

```
>>> modname = 'string'
>>> string = __import__(modname)
>>> string
<module 'string' from 'F:\\Python37\\lib\\string.py'>
```

В главе 22 также отмечалось, что более новый вызов `importlib.import_module` делает ту же самую работу, и в недавних версиях Python он, как правило, предпочтительнее для прямых вызовов импортирования по строке с именем:

```
>>> import importlib
>>> modname = 'string'
>>> string = importlib.import_module(modname)
>>> string
<module 'string' from 'F:\\Python37\\lib\\string.py'>
```

Вызов `import_module` принимает строку с именем модуля и необязательный второй аргумент, в котором указывается *пакет*, применяемый в качестве места поиска для операций относительного импортирования и по умолчанию устанавливаемый в `None`. Этот вызов работает так же, как и `__import__` в своих основных ролях; за дополнительными сведениями обращайтесь в руководства по Python.

Хотя оба вызова работают, в версиях Python, где они доступны оба, первоначальная функция `__import__` в целом предназначена для настройки операций импортирования за счет повторных присваиваний во встроенной области видимости.

Пример: транзитивная перезагрузка модулей

В настоящем разделе будет создан инструмент работы с модулями, который связывает воедино и применяет несколько ранее рассмотренных тем и служит более крупным учебным примером, подходящим для завершения главы и части. В главе 23 обсуждалась перезагрузка модулей как способ подхвата изменений в коде без остановки и перезапуска программы. Однако при перезагрузке модуля Python перезагружает только файл этого конкретного модуля; он не делает автоматической перезагрузки модулей, которые были импортированы в перезагружаемом файле.

Скажем, если вы перезагружаете модуль A, в котором импортируются модули B и C, то перезагрузка применяется только к A, но не к B и C. Операторы внутри A, которые импортируют B и C, в течение перезагрузки выполняются повторно, но они всего лишь извлекают объекты уже загруженных модулей B и C (предполагая, что они были импортированы раньше). Ниже показан действительный, хотя и абстрактный код в файле A.py:

```
# A.py
import B      # Не перезагружается при перезагрузке A!
import C      # Просто импортирует уже загруженный модуль: действия отсутствуют
% python
>>> ...
>>> from imp import reload
>>> reload(A)
```

Таким образом, по умолчанию вы не можете рассчитывать на то, что перезагрузка транзитивно подхватит изменения во всех модулях в программе — взамен вам придется использовать множество вызовов `reload` для независимого обновления подкомпонентов. В итоге интерактивное тестирование крупных систем может потребовать значительного объема работы. Вы можете проектировать свои системы так, чтобы они перезагружали подкомпоненты автоматически, добавив вызовы `reload` в родительские модули вроде A, но тогда код модулей станет сложнее.

Инструмент рекурсивной перезагрузки

Более удачный подход предусматривает написание универсального инструмента, который будет делать транзитивную перезагрузку автоматически, просматривая атрибуты `__dict__` пространств имен модулей и проверяя `type` каждого элемента, чтобы найти вложенные модули, подлежащие перезагрузке. Такая служебная функция могла бы вызывать себя *рекурсивно* для перемещения по произвольно сформированным и глубоким цепочкам зависимостей. Атрибуты модулей `__dict__` были представлены в главе 23 и задействованы ранее в текущей главе, а вызов `type` объяснялся в главе 9; нам нужно просто объединить два инструмента.

В модуле `reloadall.py`, код которого приведен ниже, определяется функция `reload_all`, автоматически перезагружающая модуль, каждый импортируемый им модуль и т.д. вплоть до самого конца каждой цепочки импортирования. Она применяет словарь для отслеживания уже перезагруженных модулей, рекурсию для прохода по цепочкам импортирования и стандартный библиотечный модуль `types`, в котором просто предопределены результаты `type` для встроенных типов. Методика со словарем `visited` позволяет здесь избежать циклов, когда операции импортирования рекурсивны или избыточны, поскольку объекты модулей неизменяемы и потому могут быть словарными ключами; как объяснялось в главах 5 и 8, множество

предложило бы аналогичную функциональность в случае использования для вставки `visited.add(module)`:

```
#!/python
"""
reloadall.py: транзитивная перезагрузка вложенных модулей (Python 2.X + 3.X).
Вызывайте reload_all с одним и более объектами импортированных модулей.
"""

import types
from imp import reload                                # from требуется в Python 3.X

def status(module):
    print('reloading ' + module.__name__)

def tryreload(module):
    try:
        reload(module)          # В Python 3.3 (только?) иногда терпело неудачу
    except:
        print('FAILED: %s' % module)

def transitive_reload(module, visited):
    if not module in visited:                         # Отлавливать циклы, дубликаты
        status(module)                               # Перезагрузить этот модуль
        tryreload(module)                          # И посетить дочерние модули
        visited[module] = True
        for attrobj in module.__dict__.values():      # Для всех атрибутов
            if type(attrobj) == types.ModuleType:       # Рекурсивный вызов,
                # если это модуль
                transitive_reload(attrobj, visited)

def reload_all(*args):
    visited = {}                                     # Главная точка входа
    for arg in args:                                # Для всех переданных аргументов
        if type(arg) == types.ModuleType:
            transitive_reload(arg, visited)

def tester(reloader, modname):                     # Код самотестирования
    import importlib, sys                          # Импортировать только при тестировании
    if len(sys.argv) > 1: modname = sys.argv[1]   # Командная строка
                                                # (или переданный аргумент)
    module = importlib.import_module(modname)     # Импортировать по строке
                                                # с именем
    reloader(module)                            # Тестируировать переданный аргумент reloader

if __name__ == '__main__':
    tester(reload_all, 'reloadall')      # Тест: перезагрузка самого себя?
```

Помимо словарей пространств имен в сценарии применяются другие исследуемые в примере инструменты. Он включает проверку `__name__` для выполнения кода самотестирования в случае запуска как сценария верхнего уровня, а его функция `tester` использует `sys.argv` для инспектирования аргументов командной строки и `importlib` для импортирования модуля по имени в строке, переданного в аргументе командной строки. Один любопытный момент: обратите внимание, что базовый вызов `reload` должен быть помещен внутрь оператора `try`, чтобы перехватывать исключения — в Python 3.3 перезагрузка иногда терпела неудачу из-за переписанных механизмов импортирования; возможно, ситуация несколько улучшится в последующих версиях (в Python 3.7 ошибку воспроизвести не удалось (исправлено?) — прим.пер.).

Предварительный обзор оператора `try` был дан в главе 10, а полностью он рассматривается в части VII.

Тестирование рекурсивной перезагрузки

Теперь, чтобы применить созданный модуль в нормальных условиях, импортируем его функцию `reload_all` и передадим ей объект уже загруженного модуля — в частности как делалось бы для встроенной функции `reload`. Когда файл запускается автономно, его код самотестирования автоматически вызывает `reload_all`, по умолчанию перезагружая собственный модуль, если аргументы командной строки не указаны. В таком режиме модуль обязан импортировать сам себя, потому что без операции импортирования его собственное имя в файле не определено. Код работает в Python 3.X и Python 2.X, поскольку при выводе мы используем + и % вместо запятой, хотя список применяемых и, следовательно, перезагружаемых модулей может варьироваться между линейками:

```
C:\code> c:\Python37\python reloadall.py
reloading reloadall
reloading types

c:\code> C:\Python27\python reloadall.py
reloading reloadall
reloading types
```

При наличии аргумента командной строки функция `tester` взамен перезагружает указанный модуль по строке с его именем — здесь модуль `pybench`, созданный в главе 21. Обратите внимание, что в этом режиме мы передаем имя модуля, а не имя файла (как и в операторах импортирования расширение .ру не включается); в конечном итоге сценарий импортирует модуль, обычным образом используя путь поиска модулей:

```
c:\code> reloadall.py pybench
reloading pybench
reloading sys
reloading os
reloading abc
reloading stat
reloading ntpath
reloading genericpath
reloading timeit
reloading gc
reloading time
reloading itertools
```

Чаще всего мы можем вводить в действие модуль `reloadall` в интерактивной подсказке — ниже демонстрируется пример для нескольких стандартных библиотечных модулей в Python 3.7 (в Python 2.X вместо `tkinter` укажите `Tkinter`):

```
>>> from reloadall import reload_all
>>> import os, tkinter
>> reload_all(os)    # Нормальный режим использования
reloading os
reloading abc
reloading sys
reloading stat
reloading ntpath
reloading genericpath
```

```
>>> reload_all(tkinter)
reloading tkinter
reloading enum
reloading sys
reloading _tkinter
reloading tkinter.constants
reloading re
reloading sre_compile
reloading _sre
reloading sre_parse
reloading functools
reloading _locale
reloading copyreg
```

Наконец, далее представлен сеанс, в котором сравнивается нормальная и транзитивная перезагрузка – изменения, внесенные в два вложенных файла, подхватываются только при транзитивной перезагрузке:

```
import b    # Файл a.py
X = 1

import c    # Файл b.py
Y = 2

Z = 3      # Файл c.py

C:\code> py -3
>>> import a
>>> a.X, a.b.Y, a.b.c.Z
(1, 2, 3)

# Не останавливая Python, измените присваиваемые значения
# во всех трех файлах и сохраните их

>>> from imp import reload
>>> reload(a)          # Встроенная функция reload выполняет
                        # перезагрузку только на верхнем уровне
<module 'a' from '.\a.py'>
>>> a.X, a.b.Y, a.b.c.Z
(111, 2, 3)
>>> from reloadall import reload_all
>>> reload_all(a)       # Нормальный режим использования
reloading a
reloading b
reloading c
>>> a.X, a.b.Y, a.b.c.Z    # Перезагружает также и все вложенные модули
(111, 222, 333)
```

Изучите код инструмента перезагрузки и результаты его запуска, чтобы больше узнать о его работе. В следующем разделе мы продолжим его развивать.

Альтернативные реализации

Для поклонников рекурсии ниже показана альтернативная *рекурсивная* версия функции из предыдущего раздела – для обнаружения циклов в ней применяется *множество*, она чуть более *прямая* из-за устранения цикла верхнего уровня и служит общей иллюстрацией методик с рекурсивными функциями (сравните ее с первоначальной версией, чтобы выяснить отличия). Данная версия получает часть работы даром от первоначальной, хотя порядок перезагрузки модулей может меняться, если также меняется порядок в словарях пространств имен:

```

"""
reloadall2.py: транзитивная перезагрузка вложенных модулей
(альтернативная версия)
"""

import types
from imp import reload          # from требуется в Python 3.X
from reloadall import status, tryreload, tester

def transitive_reload(objects, visited):
    for obj in objects:
        if type(obj) == types.ModuleType and obj not in visited:
            status(obj)
            tryreload(obj)           # Перезагрузить это, рекурсия по атрибутам
            visited.add(obj)
            transitive_reload(obj.__dict__.values(), visited)

def reload_all(*args):
    transitive_reload(args, set())

if __name__ == '__main__':
    tester(reload_all, 'reloadall2')  # Тест: перезагрузка самого себя?

```

Как объяснялось в главе 19, для большинства рекурсивных функций обычно имеется эквивалент в виде *явного стека* или очереди, который в ряде контекстов может оказаться предпочтительнее. Далее представлен инструмент транзитивной перезагрузки такого рода; в нем используется генераторное выражение для фильтрации объектов, отличающихся от модулей, и модулей, уже посещенных в пространстве имен текущего модуля. По причине извлечения и добавления элементов в конец списка он основан на стеке, хотя порядок заталкивания элементов и значения словаря влияют на порядок, в котором достигаются и перезагружаются модули. Инструмент посещает подмодули в словарях пространств имен справа налево в отличие от порядка слева направо, поддерживаемого в рекурсивных версиях. Мы могли бы это изменить, но порядок в словарях все равно произвольный.

```

"""
reloadall3.py: транзитивная перезагрузка вложенных модулей (явный стек)
"""

import types
from imp import reload          # from требуется в Python 3.X
from reloadall import status, tryreload, tester

def transitive_reload(modules, visited):
    while modules:
        next = modules.pop()          # Удалить элемент next в конце
        status(next)                  # Перезагрузить это, затолкнуть атрибуты в стек
        tryreload(next)
        visited.add(next)
        modules.extend(x for x in next.__dict__.values())
        if type(x) == types.ModuleType and x not in visited

def reload_all(*modules):
    transitive_reload(list(modules), set())

if __name__ == '__main__':
    tester(reload_all, 'reloadall3')  # Тест: перезагрузка самого себя?

```

Если рекурсивная и нерекурсивная версии сбивают с толку, тогда еще раз просмотрите обсуждение рекурсивных функций в главе 19.

Тестирование версий инструмента перезагрузки

Чтобы удостовериться в том, все три версии инструмента перезагрузки работают одинаково, давайте протестируем их. Благодаря общей функции тестирования мы можем запускать все три версии в командной строке без аргументов для проверки перезагрузки самого модуля и с именем модуля, подлежащего перезагрузке (в `sys.argv`):

```
c:\code> reloadall.py
reloading reloadall
reloading types

c:\code> reloadall2.py
reloading reloadall2
reloading types

c:\code> reloadall3.py
reloading reloadall3
reloading types
```

Несмотря на то что это трудно заметить, мы на самом деле протестировали альтернативные версии инструмента перезагрузки – в каждом teste применяется общая функция `tester`, но ей передается `reload_all` из собственного файла. Далее демонстрируется перезагрузка всеми версиями модуля `tkinter` из Python 3.X и всех импортируемых в нем модулей:

```
c:\code> reloadall.py tkinter
reloading tkinter
reloading enum
reloading sys
reloading _tkinter
reloading tkinter.constants
reloading re
reloading sre_compile
reloading _sre
reloading sre_parse
reloading functools
reloading _locale
reloading copyreg

c:\code> reloadall2.py tkinter
reloading tkinter
reloading enum
reloading sys
reloading _tkinter
reloading tkinter.constants
reloading re
reloading sre_compile
reloading _sre
reloading sre_parse
reloading functools
reloading _locale
reloading copyreg

c:\code> reloadall3.py tkinter
reloading tkinter
reloading re
reloading copyreg
reloading _locale
reloading functools
```

```
reloading sre_parse
reloading sre_compile
reloading _sre
reloading enum
reloading sys
reloading tkinter.constants
reloading _tkinter
reloading sys
reloading enum
```

Все три версии инструмента перезагрузки работают в Python 3.X и 2.X – они обеспечивают унифицированный вывод и избегают использования средств, специфичных для версии (хотя в случае Python 2.X потребуется указать имя модуля Tkinter):

```
c:\code> py -2 reloadall.py
reloading reloadall
reloading types

c:\code> py -2 reloadall2.py Tkinter
reloading Tkinter
reloading _tkinter
reloading FixTk
...и так далее...
```

Как обычно, мы можем также выполнять тестирование интерактивно, импортируя и вызывая либо главную точку входа с объектом модуля, либо функцию тестирования с передачей ей функций перезагрузки и строки с именем модуля:

```
C:\code> py -3
>>> import reloadall, reloadall2, reloadall3
>>> import tkinter
>>> reloadall.reload_all(tkinter)           # Нормальное использование
reloading tkinter
reloading enum
reloading sys
...и так далее...
>>> reloadall.tester(reloadall2.reload_all, 'tkinter')  # Функция тестирования
reloading tkinter
reloading enum
reloading sys
...и так далее...
>>> reloadall.tester(reloadall3.reload_all, 'reloadall3') # Имитация кода
                                                               # самотестирования
reloading reloadall3
reloading types
```

Наконец, если вы заглянете в показанный ранее вывод перезагрузок `tkinter`, то можете заметить, что все три версии могут выдавать результаты в отличающемся порядке; все они зависят от упорядочения словарей пространств имен, а последние две полагаются на порядок, в котором элементы добавляются в их стек. Фактически в Python 3.3, например, порядок перезагрузки отдельной версии инструмента мог даже варьироваться от запуска к запуску. Чтобы удостовериться в том, что все три версии перезагружают те же самые модули безотносительно к порядку, в котором они это делают, мы можем применить множества (или сортировку) для реализации нейтральной к порядку проверки выводимых сообщений на предмет эквивалентности (сообщения здесь получаются за счет запуска команд оболочки с помощью вызова `os.popen`, который встречался в главах 18 и 21):

```
>>> import os
>>> res1 = os.popen('reloadall.py tkinter').readlines()
>>> res2 = os.popen('reloadall2.py tkinter').readlines()
>>> res3 = os.popen('reloadall3.py tkinter').readlines()
>>> res1[:3]
['reloading tkinter\n', 'reloading enum\n', 'reloading sys\n']
>>> res1 == res2, res2 == res3
(True, False)
>>> set(res1) == set(res2), set(res2) == set(res3)
(True, True)
```

Запустите сценарии, изучите их код и поэкспериментируйте с ними для более глубокого понимания; они представляют собой импортируемые инструменты, который вы, возможно, пожелаете добавить в свою библиотеку исходного кода. Похожая методика тестирования используется при рассмотрении инструментов для построения листингов деревьев классов в главе 31, где мы будем применять ее для передачи объектов классов, расширив еще больше.

Также имейте в виду, что все три версии перезагружают только модули, которые были загружены с помощью операторов `import` — поскольку имена, копируемые посредством операторов `from`, не приводят к вложению модуля и помещению ссылки на него в пространство имен импортера, содержащий имена модуль не перезагружается. По существу инструменты транзитивной перезагрузки опираются на тот факт, что перезагрузки модулей обновляют объекты модулей *на месте*, поэтому все ссылки на такие модули в любой области видимости автоматически будут видеть обновленную версию. Из-за того, что импортеры `from` копируют имена, они не обновляются за счет перезагрузки — транзитивной или нет — и поддержка такого обновления может требовать либо анализа исходного кода, либо настройки работы импортирования (указания ищите в главе 22).

Подобного рода воздействия со стороны инструментов, вероятно, являются еще одной причиной отдавать предпочтение оператору `import` перед `from`, что подводит нас к концу главы и части, а также к стандартному набору предостережений по теме настоящей части.

Затруднения, связанные с модулями

В этом разделе мы рассмотрим обычный набор граничных сценариев, которые могут сделать жизнь гораздо интереснее для тех, кто начинает программировать на Python. Некоторые из них приводятся здесь, а некоторые настолько неочевидны, что их трудно сопроводить характерными примерами, но большинство сценариев иллюстрируют какой-то важный аспект языка.

Конфликты имен модулей: операции импортирования пакетов и относительно пакетов

При наличии двух модулей с тем же самым именем вы можете импортировать только один из них — по умолчанию всегда будет выбираться модуль, чей каталог оказывается крайним слева в пути поиска модулей `sys.path`. Это не проблема, если предпочитаемый модуль находится в каталоге вашего сценария верхнего уровня; поскольку он всегда будет первым в пути поиска модулей, поиск производится сначала в его содержимом. Тем не менее, для операций импортирования между каталогами

линейная природа пути поиска модулей означает, что файлы с одинаковыми именами могут конфликтовать.

Для исправления либо избегайте одинаково именованных файлов, либо используйте средство импортирования пакетов из главы 24. Если вам нужно получить доступ к обоим файлам с одинаковыми именами, тогда поместите свои файлы исходного кода в подкаталоги, так чтобы имена каталогов при импортировании пакетов делали ссылки на модули уникальными. До тех пор, пока имена каталогов включающих пакетов уникальны, вы будете иметь возможность доступа к любому или обоим одинаково именованным модулям.

Обратите внимание, что такая проблема может также возникать, если вы случайно выбираете для своего модуля имя, совпадающее с именем стандартного библиотечного модуля, который вам необходим. Дело в том, что ваш локальный модуль в домашнем каталоге программы (или в другом каталоге, находящемся раньше в пути поиска модулей) может скрыть и заменить библиотечный модуль.

Для исправления либо избегайте применять для модуля такое же имя, как у другого необходимого вам модуля, либо храните свои модули в каталоге пакета и используйте модель импортирования относительно пакетов Python 3.X, доступную как вариант в Python 2.X. В рамках такой модели нормальные операции импортирования пропускают поиск в каталоге пакета (так что вы получите версию из библиотеки), но специальные операторы импортирования с точками по-прежнему способны выбирать локальную версию модуля, если она нужна.

Порядок следования операторов в коде верхнего уровня имеет значение

Как вам уже известно, когда модуль импортируется в первый раз (или перезагружается), Python выполняет его операторы друг за другом от начала файла до его конца. В результате возникает несколько тонких последствий в отношении ссылок вперед, которые полезно подчеркнуть здесь.

- Код на *верхнем уровне* файла модуля (не вложенный в какую-нибудь функцию) выполняется, как только Python его достигает во время выполнения операции импортирования; по этой причине в нем нельзя ссылаться на имена, присваиваемые *далее* в файле.
- Код внутри тела какой-то *функции* не запускается до тех пор, пока функция не будет вызвана; поскольку имена в функции не распознаются вплоть до ее действительного выполнения, внутри функции обычно можно ссылаться на имена, находящиеся где угодно в файле.

Как правило, ссылки вперед становятся проблемой только в коде модуля верхнего уровня, который выполняется немедленно; функции могут ссылаться на имена произвольным образом. В следующем файле иллюстрируются правила ссылок вперед:

```
func1()           # Ошибка: имя func1 пока еще не присвоено
def func1():
    print(func2())
# Нормально: имя func2 ищется позже
func1()           # Ошибка: имя func2 пока еще не присвоено
def func2():
    return "Hello"
func1()           # Нормально: имена func1 и func2 присвоены
```

Когда данный файл импортируется (или запускается как автономная программа), Python выполняет его операторы с начала до конца. Первый вызов `func1` терпит неудачу, т.к. оператор `func1 def` пока еще не выполнялся. Вызов `func2` внутри `func1` работает при условии, что оператор `def` для `func2` был достигнут ко времени вызова `func1` – и это не происходит к моменту выполнения второго вызова `func1`. Последний вызов `func1` в конце файла работает, потому что имена `func1` и `func2` оба присвоены.

Смешивание операторов `def` с кодом верхнего уровня не только затрудняет его чтение, но также делает его зависимым от порядка следования операторов. В качестве эмпирического правила запомните: если вам необходимо смешивать немедленно выполняющийся код с операторами `def`, тогда размещайте операторы `def` в начале файла, а следом за ними код верхнего уровня. В таком случае ваши функции гарантированно будут определены и присвоены ко времени, когда Python начнет выполнять код, где они применяются.

Оператор `from` копирует имена, но не ссылки на них

Несмотря на широкое применение, оператор `from` является источником разнообразных потенциальных затруднений в Python. Как вы уже знаете, оператор `from` в действительности представляет собой присваивание именам в области видимости импортера – операцию копирования имени, а не создание псевдонима имени. Последствия такие же, как для всех присваиваний в Python, но они едва различимы, особенно учитывая, что код, который совместно использует объекты, находится в разных файлах. Например, пусть мы определили приведенный ниже модуль `nested1.py`:

```
# nested1.py
X = 99
def printer(): print(X)
```

Если мы импортируем два его имени с применением оператора `from` в другом модуле, `nested2.py`, то получим копии этих имен, а не ссылки на них. Изменение имени в импортере переустанавливает только привязку локальной версии данного имени, но не имя в `nested1.py`:

```
# nested2.py
from nested1 import X, printer      # Копировать имена
X = 88                                # Изменяет только X в этом модуле!
printer()                               # X в nested1 по-прежнему 99
% python nested2.py
99
```

Однако если мы используем оператор `import` для получения модуля целиком и затем присваиваем уточненному имени, тогда изменяется имя в `nested1.py`. Уточнение посредством атрибутов направляет Python на имя в объекте модуля, а не на имя в импортере `nested3.py`:

```
# nested3.py
import nested1    # Получить модуль как единое целое
nested1.X = 88    # Нормально: изменяет X в nested1
nested1.printer()
% python nested3.py
88
```

Форма оператора `from *` Может сделать неясным смысл переменных

Я упоминал об этом ранее, но отложил изложение деталей до настоящего момента. Поскольку в случае применения формы оператора `from module import *` вы не указываете список желаемых переменных, возможно случайное перезаписывание имен, которые уже используются в текущей области видимости. Хуже того, может затрудниться определение, откуда поступает переменная. Это особенно верно, когда форма `from *` применяется для более чем одного импортируемого файла.

Например, если вы используете `from *` для трех модулей, как показано далее, то у вас не будет возможности узнать, какая на самом деле будет вызвана функция, разве что произвести поиск во всех трех внешних файлах модулей, которые вполне могут располагаться в других каталогах:

```
>>> from module1 import *    # Неудачно: может молча перезаписать текущие имена
>>> from module2 import *    # Еще хуже: нет способа выяснить, что мы получаем!
>>> from module3 import *
>>> ...
>>> func()                  # Что за функция???
```

И снова решение заключается в том, чтобы не поступать так: старайтесь явно перечислять желаемые атрибуты в операторах `from` и ограничивайте форму `from *` самое большое одним импортируемым модулем на файл. В итоге любые неопределенные имена согласно дедукции должны находиться в модуле, указанном в единственном операторе `from *`. Вы вообще можете избежать проблемы, если всегда будете применять `import` вместо `from`, но такой совет слишком строг; как и многое другое в программировании, при разумном использовании `from` является удобным инструментом. Даже приведенный пример нельзя считать абсолютно ущербным – ради удобства эту методику вполне допустимо применять в программе для сбора имен в одном месте при условии, что оно хорошо известно.

Функция `reload` может не оказывать влияния на результаты операторов импортирования `from`

Есть еще одно затруднение, связанное с `from`: как обсуждалось ранее, поскольку `from` при выполнении копирует (присваивает) имена, отсутствует связь с модулями, откуда поступили имена. Импортируемые с помощью `from` имена просто становятся ссылками на объекты, на которые ссылались такие же имена в импортируемом модуле, когда выполнялся `from`.

По причине такого поведения перезагрузка импортируемого модуля не оказывает никакого влияния на клиентов, которые импортируют его имена с использованием `from`. То есть имена клиента по-прежнему будут ссылаться на первоначальные объекты, извлеченные посредством `from`, даже если имена в исходном модуле позже переставливаются:

```
from module import X    # Имя X может не отражать результаты перезагрузки модуля!
...
from imp import reload
reload(module)          # Изменяет модуль, но не текущие имена
X                      # По-прежнему ссылается на старый объект
```

Чтобы сделать перезагрузку более эффективной, замен `from` примените `import` и уточнение имен. Поскольку уточнения всегда возвращаются к модулям, они будут об-

наруживать новые привязки к именам модулей после того, как перезагрузка обновит содержимое модуля *на месте*:

```
import module      # Получить модуль, не имена
...
from imp import reload
reload(module)      # Изменяет модуль на месте
module.X           # Получить текущее имя X: отражает перезагрузки модуля
```

В качестве связанного последствия: созданный ранее в главе инструмент транзитивной перезагрузки не применяется к именам, извлеченным с помощью `from`, а только к тем, что извлекались посредством `import`; опять-таки, если вы собираетесь использовать перезагрузки, тогда вероятно лучше придерживаться `import`.

reload, from и тестирование в интерактивном сеансе

На самом деле предыдущее затруднение является даже еще более тонким, чем кажется. В главе 3 предупреждалось, что обычно лучше не запускать программы с помощью импортирования и перезагрузок из-за присущих такому запуску сложностей. Положение дел еще больше усугубляется, когда в игру вступает также и `from`. Начинающие программисты на Python чаще всего сталкиваются с проблемами в сценариях вроде следующего – предположим, что после открытия файла модуля в окне текстового сценария вы запускаете интерактивный сеанс и тестируете свой модуль посредством `from`:

```
from module import function
function(1, 2, 3)
```

Найдя ошибку, вы переходите обратно в окно редактора, вносите изменение и пытаетесь перезагрузить модуль:

```
from imp import reload
reload(module)
```

Прием не сработает, потому что оператор `from` присваивает только имя `function`, не `module`. Чтобы сослаться на модуль в `reload`, вам придется сначала хотя бы раз привязать его имя с помощью `import`:

```
from imp import reload
import module
reload(module)
function(1, 2, 3)
```

Тем не менее, результат оказывается не совсем тем, который нужен; `reload` обновляет объект модуля *на месте*, но как обсуждалось в предыдущем разделе, имена вроде `function`, в прошлом скопированные из модуля, продолжают ссылаться на *старые объекты*; в данном случае `function` – по-прежнему исходная версия функции. Чтобы действительно получить новую функцию, после `reload` вы должны ссылаться на нее как `module.function` или повторно выполнить `from`:

```
from imp import reload
import module
reload(module)
from module import function    # Либо использовать module.function()
function(1, 2, 3)
```

Теперь наконец-то будет запускаться новая версия функции, но, похоже, для этого пришлось проделать немалую работу.

Как видите, применению `reload` вместе с `from` присущи проблемы: вы обязаны не только помнить о необходимости перезагрузки после операций импортирования, но также не забывать о повторном выполнении операторов `from` после перезагрузки. Это достаточно сложно, чтобы порой сбивать с толку даже эксперта. На самом деле ситуация в Python 3.X стала даже хуже из-за того, что вы должны также не забывать об импортировании самой функции `reload`!

Короче говоря, не стоит ожидать эффективной совместной работы `reload` и `from`. Наилучшей политикой будет отказ от их сочетания — используйте `reload` с оператором `import` или запускайте свои программы другими способами, как предлагалось в главе 3: с применением пункта меню `Run`⇒`Run Module` (Выполнить⇒Выполнить модуль) в IDLE, щелчков на значках файлов, командных строк системы или встроенной функции `exec`.

Рекурсивные операции импортирования `from` могут не работать

Я сохранил самое странное (и к счастью малоизвестное) затруднение на закуску. Поскольку операции импортирования выполняют операторы из файла от начала до конца, вы должны проявлять осмотрительность при использовании модулей, которые импортируют друг друга. Такие операции импортирования часто называют *рекурсивными*, но рекурсия в действительности не происходит (фактически здесь лучше бы подошел термин *циклические*) — операции импортирования подобного рода не увязают в бесконечных циклах импортирования. Однако из-за того, что не все операторы в модуле оказываются выполненными, когда он импортирует другой модуль, некоторые имена могут пока не существовать.

Если вы применяете `import` для получения модуля как единого целого, тогда вероятно проблема не возникнет; имена модуля не будут доступны до тех пор, пока вы не уточните их для извлечения их значений, а к этому моменту модуль, скорее всего, уже завершен. Но если вы используете `from`, чтобы извлечь специфические имена, то вы должны помнить о том, что будете иметь доступ только к тем именам из данного модуля, которые уже были присвоены, когда запустилась рекурсивная операция импортирования.

Например, рассмотрим показанные далее модули `recur1` и `recur2`. В модуле `recur1` присваивается имя `X` и затем перед присваиванием имени `Y` импортируется модуль `recur2`. В этой точке `recur2` может извлечь `recur1` как единое целое с помощью оператора `import` — он уже существует во внутренней таблице модулей Python, которая делает его импортируемым и также препятствует зацикливанию операций импортирования. Но если в модуле `recur2` применяется `from`, то он будет в состоянии видеть только имя `X`; имя `Y`, которое присваивается позже оператора `import` в `recur1`, еще не существует, а потому вы получите ошибку:

```
# recur1.py
X = 1
import recur2 # Запустить модуль recur2 прямо сейчас, когда он не существует
Y = 2

# recur2.py
from recur1 import X    # Нормально: имя X уже присвоено
from recur1 import Y    # Ошибка: имя Y пока не существует

C:\code> py -3
Traceback (most recent call last):
```

```
File "<pyshell#0>", line 1, in <module>
    import recur1
File "C:\Code\recurl.py", line 2, in <module>
    import recur2    # Запустить модуль recur2 сейчас, когда он не существует
File "C:\Code\recur2.py", line 2, in <module>
    from recurl import Y    # Ошибка: имя Y пока не существует
ImportError: cannot import name 'Y' from 'recurl' (C:\Code\recurl.py)
Трассировка (самый последний вызов указан последним):
Файл "<pyshell#0>", строка 1, в <модуль>
    import recur1
Файл "C:\Code\recurl.py", строка 2, in <модуль>
    import recur2    # Запустить модуль recur2 сейчас, когда он не существует
Файл "C:\Code\recur2.py", строка 2, in <модуль>
    from recurl import Y    # Ошибка: имя Y пока не существует
Ошибка импортирования: не удалось импортировать имя Y из recurl
(C:\Code\recurl.py)
```

Python избегает повторного выполнения операторов модуля `recurl`, когда он рекурсивно импортируется из `recur2` (иначе операции импортирования отправляли бы сценарий в бесконечный цикл, который мог бы требовать нажатия `<Ctrl+C>` или чего-то более радикального), но пространство имен модуля `recurl` является незавершенным, когда он импортируется в модуле `recur2`.

Каково решение? Не используйте `from` в рекурсивных операциях импортирования (правда, не используйте!). В противном случае, хотя Python не допустит зацикливания, ваши программы снова станут зависимыми от порядка следования операторов в модулях. В действительности есть два способа выйти из этого затруднительного положения.

- Обычно вы всегда можете устраниТЬ циклы при импортировании вроде продемонстрированного выше за счет аккуратного проектирования — доведение до максимума сцепления и минимизация связности модулей будут хорошиими начальными шагами.
- Если вы не в состоянии полностью разорвать циклы, тогда отложите доступ к именам модулей с применением операторов `import` и уточнения с помощью атрибутов (вместо `from` и прямых имен) или запускайте операторы `from` либо внутри функций (вместо верхнего уровня модуля), либо ближе к концу файла, чтобы отсрочить их выполнение.

Дополнительные соображения относительно этой проблемы вы найдете в упражнениях в конце настоящей главы.

Резюме

В главе был представлен обзор ряда более сложных концепций, связанных с модулями. Вы изучили методики скрытия данных, включение новых языковых средств с помощью модуля `__future__`, переменную режима использования `__name__`, транзитивные перезагрузки, импортирование по строкам с именами и т.д. Мы также исследовали и подвели итоги по проблемам, связанным с проектированием модулей, написали несколько более значительных программ и рассмотрели распространенные заблуждения, касающиеся модулей, чтобы помочь вам избегать их при написании своего кода.

В следующей главе мы начнем исследование *класса Python* – инструмента объектно-ориентированного программирования. Там будут применимы многие темы, раскрыты в последних нескольких главах: классы также существуют в модулях и пространствах имен, но они добавляют к процессу нахождения атрибутов дополнительный компонент, который называется *поиском в цепочке наследования*. Но прежде чем переходить к классам, закрепите пройденный материал, ответив на контрольные вопросы главы, и выполните упражнения для данной части книги.

Проверьте свои знания: контрольные вопросы

1. Что важно знать о переменных на верхнем уровне модуля, чьи имена начинаются с одиночного подчеркивания?
2. Что означает наличие у переменной `__name__` модуля строки "`__main__`"?
3. Если пользователь интерактивно вводит имя модуля для тестирования, то каким образом вы будете импортировать его в коде?
4. Чем изменение `sys.path` отличается от установки `PYTHONPATH` для модификации пути поиска модулей?
5. Если модуль `__future__` позволяет импортировать из будущего, то можно ли импортировать также из прошлого?

Проверьте свои знания: ответы

1. Переменные на верхнем уровне модуля, чьи имена начинаются с одиночного подчеркивания, *не* копируются в область видимости импортера, когда используется форма оператора `from *`. Тем не менее, они по-прежнему доступны посредством оператора `import` или нормальной формы оператора `from`. Список `__all__` похож, но является логической противоположностью; он содержит только те имена, которые *копируются* при выполнении `from *`.
2. Если значением переменной `__name__` модуля является строка "`__main__`", то это означает, что файл выполняется как сценарий верхнего уровня, а не импортируется в другом файле внутри программы. То есть файл эксплуатируется как программа, не как библиотека. Такая переменная режима использования поддерживает код с двойным режимом и код самотестирования.
3. Вводимые пользователем данные обычно поступают внутрь сценария в виде строки; чтобы импортировать модуль, имя которого указано в строке, вы можете сформировать и запустить оператор `import` с помощью `exec` или передать строку с именем в вызове `__import__` либо `importlib.import_module`.
4. Изменение `sys.path` воздействует только на одну выполняющуюся программу (процесс), и оно временно – когда программа заканчивает работу, изменение утрачивается. Настройка `PYTHONPATH` существует в среде операционной системы – она подхватывается глобально всеми вашими программами на компьютере и потому эта настройка остается после завершения программ.
5. Нет, импортировать из прошлого в Python нельзя. Мы можем установить (или упорно использовать) более старую версию языка, но обычно наилучшим выбором будет самая последняя версия Python (во всяком случае, внутри линеек – посмотрите только на долгожительство Python 2.X!).

Проверьте свои знания: упражнения для части V

Решения упражнений находятся в приложении.

1. *Основы операций импортирования.* Напишите программу, которая подсчитывает строки и символы в файле (в какой-то мере схожую по духу с утилитой `wc` в Unix). С помощью своего текстового редактора создайте модуль Python по имени `tymod.py`, который экспортирует три имени верхнего уровня.
 - о Функция `countLines(name)` читает входной файл и подсчитывает количество строк в нем (подсказка: `file.readlines` выполняет большую часть работы, а `len` делает остаток, хотя вы могли бы подсчитывать посредством `for` и файловых итераторов, чтобы поддерживать также крупные файлы).
 - о Функция `countChars(name)` читает входной файл и подсчитывает количество символов в нем (подсказка: `file.read` возвращает одиночную строку, которую можно использовать аналогичными способами).
 - о Функция `test(name)` вызывает обе функции подсчета с заданным именем входного файла. Такое имя файла обычно может передаваться, быть жестко закодированным, вводиться с помощью встроенной функции `input` либо извлекаться из командной строки через список `sys.argv`, как было показано в примерах `formats.py` и `reloadall.py` этой главы; пока что можете принять, что имя файла передается как аргумент функции.

Все три функции модуля `tymod` должны ожидать передачи строки с именем файла. Если каждая функция занимает у вас более двух или трех строк, то вы делаете чрезмерную работу – воспользуйтесь предложенными мною подсказками!

Далее протестируйте модуль в интерактивном сеансе, применяя `import` и ссылки на атрибуты для обращения к экспортимым именам. Должна ли переменная среды `PYTHONPATH` включать в себя каталог, где вы создали файл `tymod.py`? Попробуйте запустить модуль на самом себе: скажем, `test("tymod.py")`. Обратите внимание, что `test` открывает файл дважды; при должной целеустремленности вы можете суметь усовершенствовать это, передавая двум функциям подсчета объект открытого файла (подсказка: `file.seek(0)` обеспечивает возврат в начало файла).

2. *from/from *.* Протестируйте модуль `tymod` из упражнения 1 в интерактивном сеансе, используя `from` для загрузки экспортимых имен напрямую – сначала по имени, а затем с помощью формы оператора `from *` для извлечения их всех.
3. *`__main__`.* Добавьте в модуль `tymod` строку, которая вызывает функцию `test` автоматически, когда модуль запускается как сценарий, но не в случае его импортирования. По всей видимости, в добавленной строке вы будете проверять значение `__name__` на предмет равенства строке "`__main__`", как было показано в этой главе. Попробуйте запустить модуль из командной строки системы; затем импортируйте модуль и протестируйте его функции в интерактивном сеансе. Работает ли модуль в обоих режимах?
4. *Вложенные операции импортирования.* Создайте второй модуль, `myclient.py`, который импортирует модуль `tymod` и тестирует его функции; запустите `myclient` из командной строки системы. Если `myclient` применяет `from` для извлечения имен из `tymod`, то будут ли доступны функции `tymod` на верхнем уровне `myclient`? Что если взамен он будет импортировать посредством `import`?

Попробуйте написать код обеих вариаций в `myclient` и проведите тестирование в интерактивном сеансе, импортируя модуль `myclient` и инспектируя его атрибут `__dict__`.

5. *Операции импортирования пакетов.* Импортируйте свой файл из пакета. Создайте подкаталог по имени `туркг` внутри каталога из пути поиска импортируемых модулей, скопируйте или переместите в новый подкаталог файл модуля `тумод.py`, созданный в упражнении 1 или 3, и попробуйте импортировать его с помощью операции импортирования пакета в форме `import туркг.тумод`, после чего вызовите функции модуля. Попробуйте извлечь функции подсчета и посредством `from` тоже.

Чтобы все заработало, потребуется добавить файл `__init__.py` в подкаталог, куда был перемещен ваш файл модуля, но результат должен работать на всех основных платформах Python (это одна из причин использования точки в качестве разделителя внутри путей в Python). Создаваемым подкаталогом пакета может быть просто подкаталог внутри вашего рабочего каталога; в таком случае он будет найден через компонент домашнего каталога в пути поиска и вам не придется конфигурировать путь. Поместите в `__init__.py` какой-нибудь код и посмотрите, выполняется ли он в течение каждой операции импортирования.

6. *Перезагрузки.* Поэкспериментируйте с перезагрузками модуля: выполните тесты из примера `changer.py` главы 23, периодически изменяя сообщение и/или поведение вызываемой функции без остановки интерпретатора Python. В зависимости от вашей системы вы можете быть в состоянии редактировать модуль `changer` в другом окне или приостанавливать интерпретатор Python и редактировать в том же самом окне (в Unix текущий процесс обычно приостанавливают нажатием клавиатурной комбинации `<Ctrl+Z>` и возобновляют его работу посредством команды `fg`, хотя окно текстового редактора, вероятно, будет работать с тем же успехом).
7. *Циклические операции импортирования.* В разделе о затруднениях, связанных с рекурсивными (или циклическими) операциями импортирования, импортирование `recur1` приводило к возникновению ошибки. Но если вы перезапустите Python и импортируете `recur2` в интерактивном сеансе, тогда ошибка не возникает; проверьте это и убедитесь самостоятельно. Как вы думаете, почему импортирование `recur2` нормально работает, а импортирование `recur1` – нет? (Подсказка: Python сохраняет новые модули во встроенной таблице `sys.modules` – словаре – перед выполнением их кода; последующие операции импортирования сначала извлекают модуль из упомянутой таблицы, “завершен” модуль или еще нет.) Теперь попробуйте запустить `recur1` как сценарий верхнего уровня: `python recur1.py`. Получите ли вы ту же самую ошибку, которая возникает при импортировании `recur1` в интерактивном сеансе? Почему? (Подсказка: когда модули запускаются как программы, они не импортируются, поэтому данный случай дает тот же эффект, что и импортирование `recur2` в интерактивном сеансе; `recur2` является первым импортированным модулем.) Что происходит, когда вы запускаете `recur2` как сценарий? Циклические операции импортирования на практике встречаются редко. С другой стороны, если вы сумеете понять, почему они представляют собой потенциальную проблему, то узнаете много нового о семантике импортирования Python.

Решения упражнений, приводимых в конце частей

Часть I, “Начало работы”

Упражнения приведены в главе 3.

1. **Взаимодействие.** Предполагая надлежащую конфигурацию Python, взаимодействие должно выглядеть примерно так; вы можете запускать это любым желаемым способом (в IDLE, в командной оболочке и т.д.):

```
% python
... строки с информацией об авторских правах...
>>> "Hello World!"
'Hello World!'
>>>      # Для выхода используйте <Ctrl+D> либо <Ctrl+Z> или закройте окно
```

2. **Программы.** Ваш файл кода (т.е. модуля) module1.py и взаимодействие с командной оболочкой операционной системы должны быть похожи на показанные ниже:

```
print('Hello module world!')
% python module1.py
Hello module world!
```

Опять-таки при желании можете запускать файл другими способами — щелкая на значке файла, используя пункт меню Run⇒Run Module (Выполнить⇒Выполнить модуль) в IDLE и т.д.

3. **Модули.** Следующее взаимодействие иллюстрирует выполнение модуля за счет его импортирования:

```
% python
>>> import module1
Hello module world!
>>>
```

Не забывайте о том, что вам понадобится перезагрузить модуль, чтобы выполнить его снова без остановки и перезапуска интерпретатора. Вопрос о перемещении файла в другой каталог и о его повторном импортировании содержит в себе подвох. Если Python сгенерировал файл module1.rus в первоначальном каталоге, тогда он использует его при импортировании модуля, даже когда файл исходного кода (.py) был перемещен в каталог, отсутствующий в пути поиска модулей Python. Файл .rus создается автоматически, если Python имеет доступ в каталог файла исходного кода; он содержит скомпилированную версию модуля в виде байт-кода. За дополнительными сведениями о модулях обращайтесь в главу 3.

- 4. Сценарии.** Исходя из предположения, что ваша платформа поддерживает трюк с символами #!, решение будет похожим на то, что представлено далее (хотя строка #! у вас может потребовать указания другого пути). Обратите внимание, что эти строки являются значащими в случае применения запускающего модуля для Windows, который поставляется и устанавливается, начиная с версии Python 3.3, где они обеспечивают выбор версии Python для выполнения сценария, наряду со стандартной настройкой; детали и примеры ищите в приложении Б в томе 2.

```
#!/usr/local/bin/python      (или #!/usr/bin/env python)
print('Hello module world!')
% chmod +x module1.py
```

```
% module1.py
Hello module world!
```

- 5. Ошибки и отладка.** Следующее взаимодействие (запущенное в Python 3.X) демонстрирует разновидности сообщений об ошибках, которые вы будете получать при выполнении этого упражнения. В действительности вы инициируете исключения Python; стандартное поведение обработки исключений предусматривает прекращение работы выполняемой программы Python и вывод на экран сообщения об ошибке вместе с трассировкой стека. Трассировка стека показывает, где вы находились в программе, когда возникло исключение (если вызовы функций активны по время возникновения ошибки, то в разделе Traceback отображаются все уровни активных вызовов). В главе 10 и части VII вы узнаете, как перехватывать исключения с использованием операторов try и обрабатывать их произвольным образом; там вы также увидите, что Python включает полнофункциональный отладчик исходного кода для особых требований к обнаружению ошибок. А пока имейте в виду, что когда возникают программные ошибки, Python выдает значащие сообщения вместо молчаливого аварийного отказа:

```
% python
>>> 2 ** 500
32733906078961418700131896968275991522166420460430647894832913680961337
96404674554883270092325904157150886684127560071009217256545885393053328
527589376
>>>
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
Ошибка деления на ноль: деление на ноль
>>>
>>> spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в <модуль>
Ошибка в имени: имя spam не определено
```

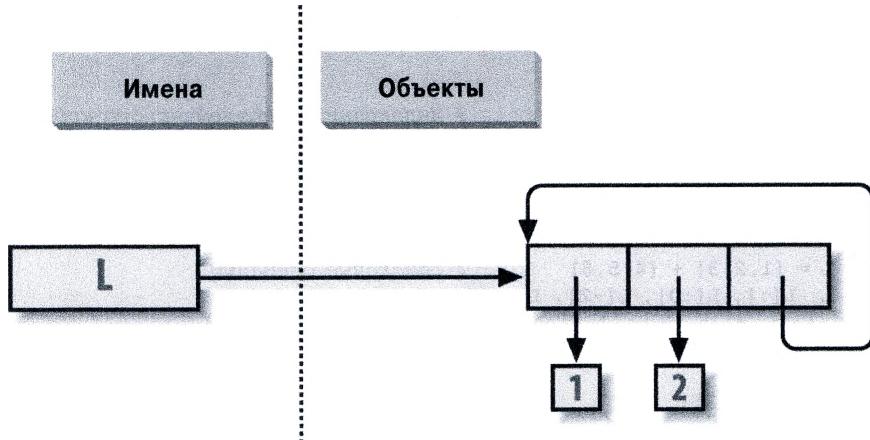
6. Прерывание работы и циклы. Когда вы набираете следующий код:

```
L = [1, 2]
L.append(L)
```

то создаете циклическую структуру данных в Python. В выпусках Python, предшествующих 1.5.1, механизм вывода Python не был настолько интеллектуальным, чтобы обнаруживать циклы в объектах. В итоге выводился бы бесконечный поток [1, 2, [1, 2, [1, 2, [1, 2 и т.д. до тех пор, пока вы не нажмете клавиатурную комбинацию прекращения (которая формально инициирует исключение, выводящее стандартное сообщение). Начиная с версии Python 1.5.1, механизм вывода достаточно искусен, чтобы обнаруживать циклы и выводить [...], тем самым уведомляя вас о том, что в структуре объекта выявлен цикл, и избегая бесконечного вывода.

Причина возникновения цикла довольно тонкая и требует знания информации, которую вы начнете получать в части II, так что здесь дается что-то вроде предварительного обзора. Но формулируя кратко, присваивания в Python всегда генерируют *ссылки* на объекты, а не их копии. Вы можете думать об объектах как об участках памяти и о ссылках как об указателях, неявно отслеживающих эти участки. Когда вы запускаете первое из представленных выше присваиваний, имя L становится именованной ссылкой на двухэлементный списковый объект – указатель на участок памяти. Списки Python на самом деле являются массивами ссылок на объекты с методом append, который изменяет массив на месте, присоединяя в конце ссылку на еще другой объект. Здесь вызов append добавляет ссылку на начало L в конец L, что приводит к циклу, проиллюстрированному на рис. Г.1: указатель в конце списка, который указывает обратно на начало списка.

Как вы узнаете в главе 6, помимо вывода особым образом циклические объекты должны также специальным способом обрабатываться сборщиком мусора Python или же занимаемое ими пространство останется невозвращенным обратно, даже когда они больше не используются. Хотя это и редко встречается на практике, в некоторых программах, которые обходят произвольные объекты или структуры, возможно, вам придется обнаруживать такие циклы самостоятельно, отслеживая места, где вы были, чтобы избежать зацикливания. Верите или нет, но циклические данные временами могут оказываться полезными, не взирая на связанный с ними особый случай вывода.



*Рис. Г.1. Циклический объект, созданный путем дополнения списка ссылкой на самого себя.
По умолчанию Python добавляет ссылку на первоначальный список, а не копию списка*

Часть II, "Типы и операции"

Упражнения приведены в главе 9.

1. Основы. Ниже показаны результаты, которые вы должны получить, наряду с комментариями о том, что они означают. Опять-таки обратите внимание, что в некоторых из них используется ; для размещения в одной строке более одного оператора (; является разделителем операторов), а запятые строят кортежи, отображаемые в круглых скобках. Также помните о том, что результат деления / отличается в Python 2.X и Python 3.X (см. главу 5) и помещение вызовов словарных методов внутрь list требуется для отображения результатов в Python 3.X, но не в Python 2.X (см. главу 8):

```
# Числа
>>> 2 ** 16                      # 2 возводится в степень 16
65536
>>> 2 / 5, 2 / 5.0               # Целочисленное деление / делает усечение
                                    # в Python 2.X, но не в Python 3.X
(0.40000000000000002, 0.40000000000000002)
# Строки
>>> "spam" + "eggs"             # Конкатенация
'spameggs'
>>> S = "ham"
>>> "eggs " + S
'eggs ham'
>>> S * 5                        # Повторение
'hamhamhamhamham'
>>> S[:0]                         # Пустой срез в начале -- [0:0]
''                                # Пустой объект того же типа, что и нарезаемый
>>> "green %s and %s" % ("eggs", S) # Форматирование
'green eggs and ham'
```

```

>>> 'green {0} and {1}'.format('eggs', s)
'green eggs and ham'

# Кортежи
>>> ('x')[0]                                # Индексация односимвольного кортежа
'x'
>>> ('x', 'y')[1]                            # Индексация двухэлементного кортежа
'y'

# Списки
>>> L = [1, 2, 3] + [4, 5, 6]      # Списковые операции
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1, 2, 3]+[4, 5, 6])[2:4]
[3, 4]
>>> [L[2], L[3]]                      # Извлечение по смещениям; сохранение в
списке
[3, 4]
>>> L.reverse(); L                   # Метод: обращение списка на месте
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L                     # Метод: сортировка списка на месте
[1, 2, 3, 4, 5, 6]
>>> L.index(4)                      # Метод: смещение на 4 от начала (поиск)
3

# Словари
>>> {'a':1, 'b':2}['b']            # Индексация словаря по ключу
2
>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0                      # Создание нового элемента
>>> D['x'] + D['w']
1
>>> D[(1,2,3)] = 4                # Кортеж, используемый как ключ (неизменяемый)
>>> D
{'w': 0, 'z': 3, 'y': 2, (1, 2, 3): 4, 'x': 1}
>>> list(D.keys()), list(D.values()), (1,2,3) in D # Методы, проверка ключа
([{'w', 'z', 'y', (1, 2, 3), 'x'}, [0, 3, 2, 4, 1], True)

# Пустые объекты
>>> [[], ["", [], (), {}, None]]    # Множество пустых объектов
([[], ['', [], (), {}, None]])

```

2. Индексация и нарезание. Индексация за границами (например, L[4]) приводит к ошибке; Python всегда осуществляет проверку, что все смещения находятся внутри границ последовательности.

С другой стороны, нарезание за границами (скажем, L[-1000:100]) работает, поскольку Python масштабирует срезы, выходящие за границы, чтобы они всегда умещались (при необходимости пределы устанавливаются в ноль или в длину последовательности).

Извлечение последовательности в обратном порядке, когда нижняя граница больше верхней (например, L[3:1]), на самом деле не работает. Вы получите пустой срез ([]), т.к. Python масштабирует пределы нарезания, чтобы нижняя граница всегда была меньше или равна верхней границе (скажем, L[3:1] масш-

табирируется до `L[3:3]`, т.е. пустая точка вставки со смещением 3). Срезы всегда извлекаются слева направо, даже если вы указываете отрицательные индексы (они сначала преобразуются в положительные за счет добавления длины последовательности). Обратите внимание, что срезы с тремя пределами, появившиеся в Python 2.3, несколько модифицируют это поведение. Например, `L[3:1:-1]` извлекает справа налево:

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка индекса: индекс в списке вышел за допустимые пределы
>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]
>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. Индексация, нарезание и оператор `del`. Взаимодействие с интерпретатором должно выглядеть так, как показано ниже. Обратите внимание, что присваивание пустого списка по смещению сохраняет там пустой списковый объект, но присваивание пустого списка срезу удаляет этот срез. Присваивание срезу ожидает другой последовательности, иначе возникнет ошибка типа; оно вставляет элементы, имеющиеся внутри присваиваемой последовательности, а не саму последовательность:

```
>>> L = [1,2,3,4]
>>> L[2] = []
>>> L
[1, 2, [], 4]
>>> L[2:3] = []
>>> L
[1, 2, 4]
>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]
>>> L[1:2] = 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка типа: объект dict_keys не поддерживает индексацию
```

4. Присваивание кортежам. Значения X и Y меняются местами. Если кортежи появляются слева и справа от символа присваивания (=), тогда Python присваивает объекты справа целям слева в соответствии с их позициями. Вероятно, это легче понять, если отметить, что цели слева не являются настоящим кортежем, даже если выглядят похожими на него; они просто представляют собой независимые цели присваивания. Элементы справа являются кортежем, который распаковывается во время присваивания (кортеж обеспечивает временное присваивание, необходимое для достижения эффекта обмена):

```
>>> X = 'spam'  
>>> Y = 'eggs'  
>>> X, Y = Y, X  
>>> X  
'eggs'  
>>> Y  
'spam'
```

5. Ключи словарей. В качестве ключа словаря может применяться любой неизменяемый объект, в том числе целые числа, кортежи, строки и т.д. В действительности это словарь, невзирая на то, что некоторые его ключи выглядят похожими на целочисленные смещения. Ключи разнородных типов тоже нормально работают:

```
>>> D = {}  
>>> D[1] = 'a'  
>>> D[2] = 'b'  
>>> D[(1, 2, 3)] = 'c'  
>>> D  
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. Индексация словарей. Индексация по несуществующему ключу (D['d']) приводит к ошибке; присваивание по несуществующему ключу (D['d'] = 'spam') создает новый элемент словаря. С другой стороны, для списков индексация за границами также приводит к ошибке, как и присваивания за границами. Имена переменных работают подобно ключам словаря; при ссылке они должны уже быть присвоенными, но создаются во время первого присваивания. На самом деле при желании имена переменных можно обрабатывать как ключи словарей (они сделаны видимыми в пространстве имен модуля или в словарях стекового фрейма):

```
>>> D = {'a':1, 'b':2, 'c':3}  
>>> D['a']  
1  
>>> D['d']  
Traceback (innermost last):  
  File "<stdin>", line 1, in ?  
KeyError: 'd'  
Трассировка (самый последний вызов указан последним):  
  Файл <stdin>, строка 1, в ?  
Ошибка ключа: 'd'  
>>> D['d'] = 4  
>>> D  
{'b': 2, 'd': 4, 'a': 1, 'c': 3}  
>>>  
>>> L = [0, 1]
```

```

>>> L[2]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка индекса: индекс в списке вышел за допустимые пределы
>>> L[2] = 3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка индекса: присваивание в списке по индексу, выходящему за допустимые
пределы

```

7. Универсальные операции.

Ниже приведены ответы на вопросы.

- Операция + не работает на отличающихся/разнородных типах (например, строка + список, список + кортеж).
- Операция + не работает для словарей, т.к. они не являются последовательностями.
- Метод append работает только для списков, но не строк, а метод keys работает только со словарями. Метод append предполагает, что его цель представляет собой изменяемый объект, поскольку он производит расширение на месте; строки являются неизменяемыми.
- Нарезание и конкатенация всегда возвращают новый объект того же самого типа, что и обрабатываемые объекты:

```

>>> "x" + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: can only concatenate str (not "int") to str
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка типа: конкатенацию можно выполнять только строки
(не целого числа) со строкой
>>>
>>> {} + {}
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка типа: неподдерживаемый тип (типы) операнда для +: dict и dict
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: 'str' object has no attribute 'append'
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка атрибута: объект str не имеет атрибута append
>>>

```

```
>>> list({}.keys())      # list() требуется в Python 3.X, но не в Python 2.X
[]
>>> [].keys()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: 'list' object has no attribute 'keys'
Трассировка (самый последний вызов указан последним):
  Файл <stdin>, строка 1, в ?
Ошибка атрибута: объект list не имеет атрибута keys
>>>
>>> [][:]
[]
>>> """[:]"""
''
```

8. Индексация строк. Это несколько тонкий вопрос — поскольку строки являются коллекциями односимвольных строк, при каждой индексации строки вы получаете обратно строку, которую можно индексировать снова. Выражение `S[0][0][0][0][0]` просто много раз индексирует первый символ. В большинстве случаев оно не будет работать для списков (списки могут хранить произвольные объекты), если только список не содержит строки:

```
>>> S = "spam"
>>> S[0][0][0][0][0]
's'
>>> L = ['s', 'p']
>>> L[0][0][0]
's'
```

9. Неизменяемые типы. Будет работать любое из следующих двух решений. Присваивание по индексу не работает, т.к. строки неизменяемы:

```
>>> S = "spam"
>>> S = S[0] + 'l' + S[2:]
>>> S
'slam'
>>> S = S[0] + 'l' + S[2] + S[3]
>>> S
'slam'
```

(См. также строковый тип `bytearray` из Python 3.X и Python 2.6+, рассматриваемый в главе 37 — он представляет собой изменяемую последовательность небольших целых чисел, которая по существу обрабатывается так же, как строка.)

10. Вложение. Вот пример:

```
>>> me = {'name': ('John', 'Q', 'Doe'), 'age': '?', 'job': 'engineer'}
>>> me['job']
'engineer'
>>> me['name'][2]
'Doe'
```

11. Файлы. Ниже демонстрируется один из способов создания и чтения текстового файла в Python (`ls` — это команда Unix; в Windows взамен используйте `dir`):

```
# Файл: maker.py
file = open('myfile.txt', 'w')
file.write('Hello file world!\n')    # Или open().write()
file.close()                      # Вызов close не всегда обязателен
```

```

# Файл: reader.py
file = open('myfile.txt')           # 'r' - стандартный режим открытия
print(file.read())                 # Или print(open().read())

% python maker.py
% python reader.py
Hello file world!

% ls -l myfile.txt
-rwxrwxrwa 1 0      0          19 Jun 19 11:33 myfile.txt

```

Часть III, "Операторы и синтаксис"

Упражнения приведены в главе 15.

1. Написание базовых циклов. Прорабатывая это упражнение, вы в итоге получите код следующего вида:

```

>>> S = 'spam'
>>> for c in S:
...     print(ord(c))

...
115
112
97
109

>>> x = 0
>>> for c in S: x += ord(c)      # Или x = x + ord(c)

...
>>> x
433

>>> x = []
>>> for c in S: x.append(ord(c))

...
>>> x
[115, 112, 97, 109]

>>> list(map(ord, S))    # list() требуется в Python 3.X, но не в Python 2.X
[115, 112, 97, 109]
>>> [ord(c) for c in S]  # map и списковые включения автоматизируют
                           # построение списков
[115, 112, 97, 109]

```

2. Символы обратной косой черты. Пример кода 50 раз выводит символ звукового сигнала (\a); предполагая, что ваш компьютер способен обработать его и запуск производится за пределами IDLE, вы можете получить серию звуковых сигналов (или один непрерывный звук, если ваш компьютер достаточно быстрый). Словом, я вас предупредил.

3. Сортировка словарей. Ниже предлагается один из способов проработки данного упражнения (если он не совсем ясен, тогда перечитайте главу 8 или 14). Вспомните, что вы действительно должны отделять друг от друга вызовы keys и sort, поскольку sort возвращает None. В Python 2.2 и последующих версиях вы можете проходить по ключам словаря напрямую без вызова keys (скажем, for key in D:), но список ключей не будет отсортирован, как делается здесь.

В более поздних выпусках Python достичь того же эффекта можно также с помощью встроенной функции `sorted`:

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = list(D.keys())      # list() требуется в Python 3.X, но не в
Python 2.X
>>> keys.sort()
>>> for key in keys:
...     print(key, '=>', D[key])
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7
>>> for key in sorted(D):      # Лучше в более поздних выпусках Python
...     print(key, '=>', D[key])
```

4. **Альтернативные варианты программной логики.** Примеры решений показаны ниже. Для шага а присвойте результат $2^{**} X$ переменной за пределами циклов на шагах а и б, после чего используйте ее в цикле. Ваши результаты могут слегка варьироваться; данное упражнение в основном предназначено для того, чтобы дать вам возможность попрактиковаться с альтернативными вариантами кода, поэтому подойдут любые разумные версии:

```
# а
L = [1, 2, 4, 8, 16, 32, 64]
X = 5
i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print('at index', i)
        break
    i += 1
else:
    print(X, 'not found')
# б
L = [1, 2, 4, 8, 16, 32, 64]
X = 5
for p in L:
    if (2 ** X) == p:
        print((2 ** X), 'was found at', L.index(p))
        break
else:
    print(X, 'not found')
# в
L = [1, 2, 4, 8, 16, 32, 64]
X = 5
```

```

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# Г
X = 5
L = []
for i in range(7): L.append(2 ** i)
print(L)

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# е

X = 5
L = list(map(lambda x: 2**x, range(7)))      # Или [2**x for x in range(7)]
print(L)                                         # list() для вывода всего в Python
3.X, но не в Python 2.X

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

```

- 5. Сопровождение кода.** Какого-то фиксированного решения, которое можно было бы привести здесь, не существует; в качестве примера просмотрите в файле `mypydoc.py` внесенные мною правки в `pydoc.py`.

Часть IV, “Функции и генераторы”

Упражнения приведены в главе 21.

- 1. Основы.** Здесь нет ничего особенного, но обратите внимание, что использование `print` (и, следовательно, вашей функции) формально является полиморфной операцией, которая выполняет правильные действия для каждого типа объекта:

```

% python
>>> def func(x): print(x)
...
>>> func("spam")
spam
>>> func(42)
42
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'food': 'spam'})
{'food': 'spam'}

```

- 2. Аргументы.** Ниже приведен пример решения. Не забывайте, что для просмотра результатов тестовых вызовов вы должны применять `print`, поскольку код в файле отличается от кода, набираемого в интерактивной подсказке; в нормальной ситуации Python не выдает на экран результаты операторов типа выражений в файлах:

```

def adder(x, y):
    return x + y

print(adder(2, 3))
print(adder('spam', 'eggs'))
print(adder(['a', 'b'], ['c', 'd']))

% python mod.py
5
spameggs
['a', 'b', 'c', 'd']

```

3. Переменное количество аргументов. В файле `adders.py`, содержимое которого показано ниже, находятся две альтернативные версии функции `adder`. Сложная часть здесь связана с выяснением, каким образом инициализировать накопитель пустым значением любого передаваемого типа. В первом решении используется ручная проверка типа для поиска целого числа и пустой срез первого аргумента (предположительно последовательности), если аргумент определен как не являющийся целым числом. Во втором решении первый аргумент применяется для инициализации и просмотра второго и последующих элементов во многом подобно вариантам функции `min` из главы 18.

Второе решение лучше. В обоих решениях предполагается, что все аргументы относятся к тому же самому типу, и ни то, ни другое не работает со словарями (как мы узнали в части II, операция `+` не работает с разнородными типами или словарями). Вы также могли бы добавить проверку типа и специальный код, чтобы сделать возможным использование словарей, но это отдельный вопрос.

```

def adder1(*args):
    print('adder1', end=' ')
    if type(args[0]) == type(0): # Целое число?
        sum = 0                  # Инициализировать нулем
    else:                      # Иначе последовательность:
        sum = args[0][:0]       # Использовать пустой срез args[0]
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print('adder2', end=' ')
    sum = args[0]              # Инициализировать первым аргументом
    for next in args[1:]:
        sum += next            # Добавить элементы 2..N
    return sum

for func in (adder1, adder2):
    print(func(2, 3, 4))
    print(func('spam', 'eggs', 'toast'))
    print(func(['a', 'b'], ['c', 'd'], ['e', 'f']))

% python adders.py
adder1 9
adder1 spameggstoast
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 spameggstoast
adder2 ['a', 'b', 'c', 'd', 'e', 'f']

```

4. Ключевые аргументы. Далее предлагается мое решение первой и второй частей этого упражнения (содержимое файла mod.py). Для прохода по ключевым аргументам применяйте в заголовке функции форму `**args` и используйте цикл (скажем, `for x in args.keys(): use args[x]`) или применяйте `args.values()`, чтобы делать то же самое, что и суммирование позиционных аргументов `*args`:

```
def adder(good=1, bad=2, ugly=3):
    return good + bad + ugly

print(adder())
print(adder(5))
print(adder(5, 6))
print(adder(5, 6, 7))
print(adder(ugly=7, good=6, bad=5))

% python mod.py
6
10
14
18
18

# Решения второй части

def adder1(*args):                      # Суммировать любое количество
                                         # позиционных аргументов
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder2(**args):                      # Суммировать любое количество ключевых аргументов
    argskeys = list(args.keys())          # list необходим в Python 3.X!
    tot = args[argskeys[0]]
    for key in argskeys[1:]:
        tot += args[key]
    return tot

def adder3(**args):                      # То же самое, но с преобразованием в список значений
    args = list(args.values())           # list необходим для индексации в Python 3.X!
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder4(**args):                      # То же самое, но с повторным использованием
                                         # версии с позиционными аргументами
    return adder1(*args.values())

print(adder1(1, 2, 3), adder1('aa', 'bb', 'cc'))
print(adder2(a=1, b=2, c=3), adder2(a='aa', b='bb', c='cc'))
print(adder3(a=1, b=2, c=3), adder3(a='aa', b='bb', c='cc'))
print(adder4(a=1, b=2, c=3), adder4(a='aa', b='bb', c='cc'))
```

5. (И 6.) Словарные инструменты. Ниже показаны мои решения упражнений 5 и 6 (файл dicts.py). Однако это всего лишь упражнения по написанию кода, потому что в Python 1.5 появились словарные методы `D.copy()` и `D1.update(D2)` для обработки действий вроде копирования и добавления (слияния) словарей.

Ищите примеры применения `dict.update` в главе 8, а дополнительные сведения в руководстве по библиотеке Python или в книге *Python Pocket Reference* (<http://www.oreilly.com/catalog/9780596158088>). Для словарей `X[:]` не работает, т.к. они не являются последовательностями (за подробностями обращайтесь в главу 8). Кроме того, помните о том, что если вместо копирования вы присваиваете (`e = d`), то генерируете ссылку на разделяемый словарный объект; изменение `d` изменяет также и `e`:

```
def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):
    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new

% python
>>> from dicts import *
>>> d = {1: 1, 2: 2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1: 1}
>>> y = {2: 2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}
```

6. См. упражнение 5.

7. Дополнительные примеры сопоставления аргументов. Далее представлено взаимодействие, которое вы должны получить, вместе с поясняющими комментариями:

```
def f1(a, b): print(a, b)      # Нормальные аргументы
def f2(a, *b): print(a, b)     # Переменное количество позиционных аргументов
def f3(a, **b): print(a, b)    # Переменное количество ключевых аргументов
def f4(a, *b, **c): print(a, b, c)  # Смешанные режимы
def f5(a, b=2, c=3): print(a, b, c)  # Стандартные значения
def f6(a, b=2, *c): print(a, b, c)  # Стандартные значения и переменное
                                    # кол-во позиционных аргументов

% python
>>> f1(1, 2)                  # Сопоставляются по позиции (порядок имеет значение)
1 2
```

```

>>> f1(b=2, a=1)    # Сопоставляются по имени (порядок не имеет значения)
1 2

>>> f2(1, 2, 3)    # Дополнительные позиционные аргументы собираются в кортеже
1 (2, 3)

>>> f3(1, x=2, y=3) # Дополнительные ключевые аргументы собираются в словаре
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3)          # Дополнительные аргументы обоих видов
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                      # Задействованы оба стандартных значения
1 2 3

>>> f5(1, 4)                  # Используется только одно стандартное значение
1 4 3

>>> f6(1)                      # Один аргумент: соответствует a
1 2 ()

>>> f6(1, 3, 4)                # Дополнительный позиционный аргумент
                                # собирается в кортеже
1 3 (4, )

```

8. Снова простые числа. Ниже показан пример с простыми числами, помещенный внутрь функции и модуля (файл `primes.py`), так что его можно запускать много раз. Я добавил проверку `if` для отлавливания отрицательных чисел, 0 и 1. В этой версии я также применил `//` вместо `/`, чтобы сделать ее невосприимчивой к превращению операции `/` в Python 3.X в настоящее деление (см. главу 5) и обеспечить поддержку чисел с плавающей точкой (отличия в Python 2.X можно увидеть, убрав комментарий из оператора `from` и заменив `//` на `/`):

```

#from __future__ import division

def prime(y):
    if y <= 1:                                     # Для какого-то значения y > 1
        print(y, 'not prime')
    else:
        x = y // 2                                 # Операция / из Python 3.X терпит неудачу
        while x > 1:
            if y % x == 0:                         # Нет остатка от деления?
                print(y, 'has factor', x)           # Имеет сомножитель
                break                               # Пропуск else
            x -= 1
        else:
            print(y, 'is prime')                  # Является простым

prime(13); prime(13.0)
prime(15); prime(15.0)
prime(3); prime(2)
prime(1); prime(-3)

```

Далее модуль демонстрируется в действии; операция `//` позволяет его использовать также для чисел с плавающей точкой, хотя возможно поступать подобным образом не следует:

```

% python primes.py
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0

```

```
3 is prime
2 is prime
1 not prime
-3 not prime
```

Функция `prime` по-прежнему не слишком пригодна к многоократному применению (она могла бы возвращать значения, а не выводить их), но для проведения экспериментов ее вполне достаточно. Она также не выдает математически строгое простое число (работает для чисел с плавающей точкой) и не особенно эффективна. Усовершенствования оставляются в качестве упражнений для математически подкованных читателей. (Подсказка: цикл `for` по `range(y, 1, -1)` может быть чуть быстрее, чем `while`, но настоящим узким местом здесь является алгоритм.) Для измерения времени выполнения альтернативных вариантов используйте любительский модуль `timer` или стандартный библиотечный модуль `timeit` и кодовые шаблоны, которые должны быть похожи на применяемые в разделах главы 21, посвященных измерению времени (просмотрите также решение упражнения 10).

9. Итерации и включения. Вот вариант кода, который вы должны написать; правда, я предполагаю одно, но вы возможно другое:

```
>>> values = [2, 4, 9, 16, 25]
>>> import math

>>> res = []
>>> for x in values: res.append(math.sqrt(x))

...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(map(math.sqrt, values))
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(math.sqrt(x) for x in values)
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

10. Измерение времени выполнения инструментов. Ниже показан код, который я написал для измерения времени выполнения трех реализаций, вычисляющих квадратный корень, наряду с результатами, полученными в CPython 3.7 и 2.7, а также PyPy 1.9 (реализует Python 2.7). Каждый тест выбирает лучшее время из трех запусков; каждый запуск берет суммарное время, требуемое для вызова тестовой функции 1 000 раз; и каждая тестовая функция выполняет 1 000 итераций. Последний результат каждой функции выводится для проверки, что все три делают ту же самую работу:

```
# Файл timer2.py (Python 2.X и 3.X)
...содержимое такое же, как было показано в главе 21...

# Файл timesqrt.py
import sys, timer2
reps = 10000
repslist = range(reps) # Для Python 2.X
from math import sqrt # Не math.sqrt: добавляет время извлечения атрибутов
def mathMod():
```

```

for i in repslist:
    res = sqrt(i)
return res

def powCall():
    for i in repslist:
        res = pow(i, .5)
    return res

def powExpr():
    for i in repslist:
        res = i ** .5
    return res

print(sys.version)
for test in (mathMod, powCall, powExpr):
    elapsed, result = timer2.bestoftotal(test, _reps1=3, _reps=1000)
    print ('%s: %.5f => %s' % (test.__name__, elapsed, result))

```

Ниже представлены результаты выполнения в трех версиях Python. Выполнение в Python 3.7 и 2.7 оказывается быстрее, чем в версиях Python 3.0 и 2.6, рассматриваемых в предыдущем издании, из-за более производительного компьютера. Для каждой версии Python, где проводилось тестирование, модуль math работает быстрее выражения `**`, которое быстрее вызова `pow`; тем не менее, вы должны испытать свой код на своем компьютере с имеющейся версией Python. Кроме того, обратите внимание, что в этом тесте Python 3.7 медленнее, чем Python 2.7, а PyPy гораздо быстрее обеих версий CPython несмотря на наличие математики с плавающей точкой и итераций. Ситуация в более поздних версиях Python может отличаться, так что проверьте все самостоятельно:

```

c:\code> py -3 timesqrt.py
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]
mathMod: 2.55214 => 99.99499987499375
powCall: 3.80201 => 99.99499987499375
powExpr: 3.05757 => 99.99499987499375

c:\code> py -2 timesqrt.py
2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)]
mathMod: 1.04337 => 99.994999875
powCall: 2.57516 => 99.994999875
powExpr: 1.89560 => 99.994999875

c:\code> c:\pypy\pypy-1.9\pypy timesqrt.py
2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit]
mathMod: 0.07491 => 99.994999875
powCall: 0.85678 => 99.994999875
powExpr: 0.85453 => 99.994999875

```

Чтобы измерить относительные скорости выполнения *включений словарей* Python 3.X и 2.7 и эквивалентных циклов `for` в интерактивном сеансе, вы можете организовать взаимодействие вроде приведенного далее. Похоже, что в Python 3.7 скорости выполнения включений словарей и эквивалентных циклов `for` примерно одинаковы. Однако в отличие от списковых включений в настоящее время ручные циклы слегка быстрее включений словарей (хотя разница не особо значительная — в итоге при создании 50 словарей, каждый из которых состоит из 1 000 000 элементов, мы экономим около двух секунд). Как всегда, вместо

того, чтобы воспринимать эти результаты как абсолютную истину, вы должны провести самостоятельные исследования на своем компьютере и со своей версией Python:

```
C:\code> c:\python37\python
>>>
>>> def dictcomp(I):
    return {i: i for i in range(I)}
>>> def dictloop(I):
    new = {}
    for i in range(I): new[i] = i
    return new
>>> dictcomp(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>> dictloop(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>>
>>> from timer2 import total, bestof
>>> bestof(dictcomp, 10000) [0]          # Словарь из 10 000 элементов
0.001206016999987014
>>> bestof(dictloop, 10000) [0]
0.001582992999999533
>>>
>>> bestof(dictcomp, 100000) [0]         # Словарь из 100 000 элементов:
#   на порядок медленнее
0.01313573400000223
>>> bestof(dictloop, 100000) [0]
0.017253260000003934
>>>
>>> bestof(dictcomp, 1000000) [0]        # Словарь из 1 000 000 элементов:
#   еще на порядок медленнее
0.1279122199999847
>>> bestof(dictloop, 1000000) [0]
0.16962867899999878
>>>
>>> total(dictcomp, 1000000, _reps=50) [0] # Итоги по созданию 50 словарей
#   из 1 000 000 элементов
7.04316026699999
>>> total(dictloop, 1000000, _reps=50) [0]
9.104445693000002
```

11. Рекурсивные функции. Я написал эту функцию следующим образом; подошла бы также встроенная функция range, включение или map, но рекурсия достаточно полезна, чтобы поэкспериментировать с ней здесь (print является функцией в Python 3.X, если только вы не импортировали ее из модуля `_future_` или не реализовали собственный эквивалент):

```
def countdown(N):
    if N == 0:
        print('stop')          # Python 2.X: print 'stop'
    else:
        print(N, end=' ')
        countdown(N-1)
>>> countdown(5)
5 4 3 2 1 stop
```

```

>>> countdown(20)
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 stop
# Нерекурсивные варианты:
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
# Только в Python 3.X:
>>> t = [print(i, end=' ') for i in range(5, 0, -1)]
5 4 3 2 1
>>> t = list(map(lambda x: print(x, end=' '), range(5, 0, -1)))
5 4 3 2 1

```

Из соображений гуманности я не хотел включать в решение этого упражнения вариант на основе *генератора*, но все-таки он приведен ниже; в данном случае все остальные методики выглядят гораздо более простыми – хороший пример ситуации, когда лучше избегать использования генераторов. Не забывайте, что генераторы не производят результатов до прохода по ним, поэтому здесь нужен оператор `for` или `yield from` (`yield from` работает, только начиная с версии Python 3.3):

```

def countdown2(N):                                # Генераторная функция, рекурсивная
    if N == 0:
        yield 'stop'
    else:
        yield N
        for x in countdown2(N-1): yield x      # Python 3.3+:
                                                # yield from countdown2(N-1)

>>> list(countdown2(5))
[5, 4, 3, 2, 1, 'stop']

# Nonrecursive options:
>>> def countdown3():                            # Генераторная функция, более простая
                                                # До версии Python 3.3:
        yield from range(5, 0, -1)                # for x in range(): yield x

>>> list(countdown3())
[5, 4, 3, 2, 1]

>>> list(x for x in range(5, 0, -1))  # Эквивалентное генераторное выражение
[5, 4, 3, 2, 1]

>>> list(range(5, 0, -1))                  # Эквивалентная негенераторная форма
[5, 4, 3, 2, 1]

```

- 12. Вычисление факториалов.** Ниже показано мое решение этого упражнения; оно выполняется под управлением Python 3.X и 2.X, а в строковом литерале в конце приведен вывод, полученный в Python 3.7. Естественно, существует множество возможных вариаций такого кода; например, `range` можно было бы вызывать для `2..N+1`, чтобы пропустить итерацию, а в `fact2` можно было бы применять `reduce(operator.mul, range(N, 1, -1))`, избежав использования `lambda`.

```

#!python
from __future__ import print_function      # Файл factorials.py
from functools import reduce
from timeit import repeat
import math

```

```

def fact0(N):
    if N == 1:                                # Рекурсивный подход
        return N
    else:
        return N * fact0(N-1)

def fact1(N):
    return N if N == 1 else N * fact1(N-1)  # Рекурсивный подход,
                                                # односторочная реализация

def fact2(N):                                # Функциональный подход
    return reduce(lambda x, y: x * y, range(1, N+1))

def fact3(N):
    res = 1
    for i in range(1, N+1): res *= i          # Итерационный подход
    return res

def fact4(N):
    return math.factorial(N)      # "Батарейки" из стандартной библиотеки

# Тесты
print(fact0(6), fact1(6), fact2(6), fact3(6), fact4(6))  # 6*5*4*3*2*1:
                                                               # все дают 720
print(fact0(500) == fact1(500) == fact2(500) == fact3(500) == fact4(500))
# True

for test in (fact0, fact1, fact2, fact3, fact4):
    print(test.__name__, min(repeat(stmt=lambda: test(500), number=20,
repeat=3)))

"""
C:\code> py -3 factorials.py
720 720 720 720 720
True
fact0 0.006387722999999956
fact1 0.006470778000000066
fact2 0.00447288499999982
fact3 0.0037610310000000258
fact4 0.0011195480000000257
"""

```

Вывод: рекурсивная реализация на компьютере с имеющейся версией Python выполняется медленнее всех и терпит неудачу, как только N достигает значения 999, из-за стандартного размера стека, установленного в sys. В главе 19 упоминалось, что данный предел может быть увеличен, но в любом случае простые циклы или стандартный библиотечный инструмент выглядят более удачным вариантом.

Такой общий вывод часто остается справедливым. Скажем, ''.join(reversed(S)) может оказываться предпочтительным способом обращения строки, несмотря на то, что возможны рекурсивные решения. Измерьте время выполнения следующего кода, чтобы выяснить почему: что касается вычисления факториалов в Python 3.X, то рекурсия на сегодняшний день на порядок медленнее в CPython, хотя в PyPy результаты могут варьироваться:

```

def rev1(S):
    if len(S) == 1:
        return S

```

```

else:
    return S[-1] + rev1(S[:-1])      # Рекурсивная реализация:
                                    # на порядок медленнее в CPython
def rev2(S):
    return ''.join(reversed(S))      # Нерекурсивная реализация
                                    # с итерацией: проще, быстрее
def rev3(S):
    return S[::-1]                  # Даже лучше?: обращение последовательности
                                    # посредством нарезания

```

Часть V, "Модули и пакеты"

Упражнения приведены в главе 25.

1. Основы операций импортирования. После проработки упражнения итоговый файл (`mymod.py`) и взаимодействие должны выглядеть следующим образом; вспомните, что Python способен читать целый файл в список строк, астроенная функция `len` возвращает длины строк и списков:

```

def countLines(name):
    file = open(name)
    return len(file.readlines())
def countChars(name):
    return len(open(name).read())
def test(name):                      # Или передать файловый объект
    return countLines(name), countChars(name)  # Или возвратить словарь
% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291)

```

Ваши функции подсчета могут варьироваться, как и мои могут включать или нет комментарии и дополнительную строку в конце. Обратите внимание, что эти функции загружают файл в память целиком, так что они не будут работать с крайне большими файлами, не умещающимися в память компьютера. Чтобы обеспечить более высокую надежность, взамен вы могли бы читать построчно с помощью итераторов и подсчитывать по мере продвижения:

```

def countLines(name):
    tot = 0
    for line in open(name): tot += 1
    return tot
def countChars(name):
    tot = 0
    for line in open(name): tot += len(line)
    return tot

```

Такой же эффект может дать генераторное выражение (хотя инструктор может снять баллы за чрезмерную магию!):

```

def countlines(name): return sum(+1 for line in open(name))
def countchars(name): return sum(len(line) for line in open(name))

```

В среде Unix вы можете проверить вывод посредством команды `wc`; в среде Windows щелкните правой кнопкой мыши на значке файла, чтобы просмотреть

его свойства. Обратите внимание, что ваш сценарий может сообщать о меньшем количестве символов, чем среда Windows – в целях переносимости Python преобразует маркеры `\r\n` конца строки Windows в `\n`, отбрасывая тем самым 1 байт (символ) на строку. Для точного соответствия счетчиков байтов с Windows вы должны открыть файл в двоичном режиме ('`rb`') или добавить количество байтов, равное количеству строк. Дополнительные сведения о трансляции символов конца строки ищите в главах 9 и 37.

“Амбициозная” часть этого упражнения (передача файлового объекта, чтобы открывать файл только один раз) потребует применения метода `seek` встроенного файлового объекта. Он работает подобно функции `fseek` в языке С (и может внутренне вызывать ее): `seek` переустанавливает текущую позицию в файле согласно переданному смещению. После вызова `seek` будущие операции ввода-вывода станут выполняться относительно новой позиции. Для перехода в начало файла без его закрытия и повторного открытия вызовите `file.seek(0)`; все файловые методы `read` производят чтение с текущей позиции файла, так что для повторного чтения вам придется переходить в начало файла. Вот как может выглядеть описанная настройка:

```
def countLines(file):
    file.seek(0)                                     # Перейти в начало файла
    return len(file.readlines())
def countChars(file):
    file.seek(0)                                     # То же самое (при необходимости перейти в начало)
    return len(file.read())
def test(name):
    file = open(name)                               # Передать файловый объект
    return countLines(file), countChars(file)      # Открыть файл только один раз
>>> import mymod2
>>> mymod2.test("mymod2.py")
(11, 392)
```

2. `from / from *`. Ниже показана часть `from *`; чтобы сделать остальное, поменяйте `*` на `countChars`:

```
% python
>>> from mymod import *
>>> countChars("mymod.py")
291
```

3. `__main__`. При надлежащем написании кода этот файл работает в любом из двух режимов – запуск как программы или импортирование как модуля:

```
def countLines(name):
    file = open(name)
    return len(file.readlines())
def countChars(name):
    return len(open(name).read())
def test(name):                                     # Или передать файловый объект
    return countLines(name), countChars(name)      # Или возвратить словарь
if __name__ == '__main__':
    print(test('mymod.py'))
% python mymod.py
(13, 346)
```

Вероятно, именно здесь я бы начал подумывать об использовании аргументов командной строки или пользовательского ввода для предоставления имени файла, подлежащего подсчету, вместо его жесткого кодирования в сценарии (`sys.argv` более подробно рассматривается в главе 25, а ввод обсуждается в главе 10 – и примените `raw_input` в Python 2.X):

```
if __name__ == '__main__':
    print(test(input('Enter file name:'))) # Консоль (raw_input в Python 2.X)
if __name__ == '__main__':
    import sys                                # Командная строка
    print(test(sys.argv[1]))
```

4. Вложенные операции импортирования.

Вот мое решение (файл `myclient.py`):

```
from mymod import countLines, countChars
print(countLines('mymod.py'), countChars('mymod.py'))
```

```
% python myclient.py
13 346
```

Что касается остатка, то функции `mymod` доступны (т.е. импортируемы) из верхнего уровня `myclient`, поскольку `from` просто присваивает их именам в импортере (он работает, как если бы операторы `def` модуля `mymod` находились в `myclient`). Например, в другом файле может быть следующее:

```
import myclient
myclient.countLines(...)

from myclient import countChars
countChars(...)
```

Если в `myclient` вместо `from` используется `import`, тогда для того, чтобы добраться до функций модуля `mymod` из `myclient`, потребуется указывать путь:

```
import myclient
myclient.mymod.countLines(...)

from myclient import mymod
mymod.countChars(...)
```

В общем случае вы можете определять модули *накопителей*, которые импортируют все имена из других модулей, чтобы они были доступными в единственном удобном модуле. Скажем, в показанном ниже коде создаются три разных копии имени `somename` – `mod1.somename`, `collector.somename` и `__main__.somename`; изначально все три разделяют тот же самый объект целого числа и только имя `somename` существует в интерактивной подсказке в том виде как есть:

```
# Файл mod1.py
somename = 42

# Файл collector.py
from mod1 import *      # Накопить здесь множество имен
from mod2 import *      # From присваивает имена
from mod3 import *
>>> from collector import somename
```

5. Операции импортирования пакетов.

Для этого я поместил файл решения `mymod.py` упражнения 3 в каталог пакета. Далее описаны мои действия в консоли Windows по настройке каталога и файла `__init__.py`, который был обязательным до версии Python 3.3; вам понадобится интерполировать их для других плат-

форм (например, применять `cp` и `vi` вместо `copy` и `notepad`). Прием работает в любом каталоге (я использую здесь собственный каталог `code`), и часть действий можно выполнять в графическом пользовательском интерфейсе проводника.

Когда все было сделано, у меня появился подкаталог `mypkg`, содержащий файлы `__init__.py` и `mymod.py`. До того, как в Python 3.3 было введено расширение пакетов пространств имен, требовалось наличие файла `__init__.py` в `mypkg`, но не в родительском каталоге; формально `mypkg` находится в компоненте домашнего каталога внутри пути поиска модулей. Обратите внимание, что оператор `print`, помещенный в файл `__init__.py` каталога, выполняется только при его импортировании в первый раз, но не во второй; кроме того, в коде применяются неформатированные строки, чтобы избежать проблем с управляемыми символами внутри пути к файлу:

```
C:\code> mkdir mypkg
C:\code> copy mymod.py mypkg\mymod.py
C:\code> notepad mypkg\__init__.py
...поместить оператор print...
C:\code> python
>>> import mypkg.mymod
initializing mypkg
>>> mypkg.mymod.countLines(r'mypkg\mymod.py')
13
>>> from mypkg.mymod import countChars
>>> countChars(r'mypkg\mymod.py')
346
```

6. **Перезагрузки.** В этом упражнении вам просто предлагается поэкспериментировать с изменением примера `changer.py` из книги, так что показывать здесь нечего.
7. **Циклические операции импортирования.** Короткая история сводится к тому, что импортирование модуля `recur2` первым работает, поскольку рекурсивная операция импортирования затем происходит в операторе `import` внутри `recur1`, а не в операторе `from` `brecur2`.

Длинная история выглядит следующим образом: импортирование модуля `recur2` первым работает из-за того, что рекурсивная операция импортирования из `recur1` в `recur2` извлекает `recur2` как единое целое вместо получения специфических имен. Модуль `recur2` является незавершенным, когда он импортируется из `recur1`, но поскольку в нем используется оператор `import`, а не `from`, все проходит благополучно: Python находит и возвращает уже созданный объект модуля `recur2` и продолжает выполнять остаток модуля `recur1` без каких-либо затруднений. Когда импортирование `recur2` возобновляется, второй оператор `from` находит имя `Y` в модуле `recur1` (он выполнился полностью), а потому ни о каких ошибках не сообщается.

Запуск файла как *сценария* – не то же самое, что его импортирование в качестве модуля; такие случаи аналогичны выполнению первого оператора `import` или `from` из сценария в интерактивном сеансе. Например, запуск `recur1` как сценария работает, поскольку это то же, что и импортирование `recur2` в интерактивном сеансе, т.к. `recur2` является первым модулем, импортированным в `recur1`. Запуск `recur2` как сценария терпит неудачу по той же причине – это то же самое, что и выполнение его первого импортирования в интерактивном сеансе.

Предметный указатель

B

BDFL (Benevolent Dictator for Life), 55

C

CPython, 70

E

Eclipse, 115

I

IDE (Integrated Development Environment), 108

IronPython, 71

J

JSON (JavaScript Object Notation), 149

JVM (Java Virtual Machine), 71

Java, 71

K

Komodo, 115

L

LIFO (last-in-first-out), 277

M

MongoDB, 51

N

NetBeans, 115

NumPy (Numeric Python), 51; 204

O

ORM (Object relational mapper), 51

P

PEP (Python Enhancement Proposal), 53

PSF (Python Software Foundation), 53

PVM (Python Virtual Machine), 68

PyDev, 115

PyDoc, 472

PyMongo, 51

PyPy, 70; 72

PythonWin, 115

R

RIA (Rich Internet application), 50

S

ScientificPython, 51

SciPy, 51; 204

Silverlight, 71

Sphinx, 483

SQLite, 50

Stackless, 72

U

Unicode, 138; 156; 219

A

Аннотации, 580

Аргумент (параметр), 496; 541

передача аргументов, 541

распаковка аргументов, 553

в Python 2.X, 546

с передачей только по ключевым словам, 557; 567; 657

Архитектура программы Python, 687

Атрибут, 103; 580; 702

функции, 534; 582

Б

База данных

механизм баз данных SQLite, 50

программирование для баз данных, 50

система реляционных баз данных, 50

Байт-код, 44; 67

Библиотека

стандартная, 41; 688

Python, 48

Булевская проверка, 404

В

Ветвление

множественное, 396

Виртуальная машина Python (PVM), 68

Включение, 198; 595; 598

расширенный синтаксис включений для

множеств и словарей, 642

списковое, 603; 640

Вложение, 143

Время

модуль для измерения времени, 655

Выход

нейтральный к версии, 389

перенаправление потока вывода, 385

- Вызов, 497
Выражение, 168; 379
lambda, 585
базовое, 173
генераторное, 598; 601; 614; 619; 628
индексации, 132
справочное, 600
реза, 428
форматирования, 252
- Г**
- Генератор, 145; 152; 410; 595; 609; 623; 631
генерация перемешанных последовательностей, 626
объект генератора, 610; 614
функций, 598
- Д**
- Декораторы методов, 534
Деление
 классическое, 178
 настоящее, 178
 с округлением в меньшую сторону, 178
Динамическая типизация, 56; 207
Диспетчеры контекстов для файлов, 321
Документация Python, 466
 строки документации, 469
- З**
- Загрузка
 повторная, 101
Замыкание, 522
- И**
- Идентичность, 216
Импортирование, 101; 689
 абсолютное
 с полными путями, 745
 пакетов, 689
 синтаксис относительного импортирования, 744
Имя
 конфликты имен, 106
 межфайловое изменение имен, 707
 совмещение имен, 542
Индексация, 231; 232; 272
Инструмент
 distutils, 700
 PyDoc, 482
 pyjamas, 50
 Silverlight, 50
 Sphinx, 483
 итерационный, 306
- командной оболочки, 48
написания сценариев, 54
преобразования строк, 235
прохода по каталогам, 623
Интерактивная подсказка, 78
Интерпретатор, 64
Интерфейс, 161
 API, 50
PyMongo, 51
Tkinter, 49
графический пользовательский, 49
переносимый, 50
Исполняющая среда, 496
Итератор, 155; 282
 файловый, 313; 623
Итерация, 152; 438; 464; 620
 итерационный инструмент, 306
одноразовая, 638
по спискам, 271
протокол итерации, 152; 439; 442
ручная, 442; 444
- К**
- Каталог
 инструменты прохода по каталогам, 623
Класс, 522; 625; 633
Клиент
 с графическим пользовательским интерфейсом, 480
Ключ
 сортировка ключей, 150
Ключевые слова, 550
Код
 байт-код, 67
 инициализации, 705
 исходный, 67
Кодировка Unicode, 140
Кодовые точки (идентифицирующие числа), 224
Команды оболочки, 434
Комментарий, 173; 229; 467
Компилятор
 JIT, 74
Компонент
 интеграция компонентов, 50
Конкатенация, 133
Контейнер
 итерируемый, 197
Конфигурирование пути поиска, 696
Кортеж, 105; 153; 248; 288; 302; 307; 419
 именованный, 154; 335
 присваивание кортежей, 309

Л

Литерал, 128; 166
множества, 198
строковый, 221

М

Массив, 142
ассоциативный, 279
Матрица, 603
Машина PVM, 68
Метод, 135; 168
extend, 277
format, 251; 255
pop, 277
seek, 322
sort, 275
update, 283
декораторы методов, 534
именованный, 262
класса, 470
словарный, 283
items, 283
values, 283
строковый, 221; 239; 240
find, 242
format, 244; 258; 259
join, 242
replace, 241
split, 243
форматирования, 254
Микропотоки, 72
Множество, 145; 168; 195; 201
итерируемое, 201
Модель
относительного импортирования, 720
пакетов пространств имен, 720
Модуль, 89; 105; 684
pickle, 317; 318
struct, 320
timeit, 659
для измерения времени, 655
модификация изменяемых объектов
в модулях, 706
перезагрузка модулей, 714; 717
пространства имен модулей, 709
путь поиска модулей, 693
создание модулей, 702
стандартный библиотечный, 187

Н

Нарезание, 231; 272; 626
расширенное, 233

О

Область видимости, 504; 519; 639
встроенная, 511
лексическая, 505; 713
локальная, 509
Обращение последовательности, 234
Объект, 165; 208; 210
None, 332
type, 333
генераторный, 433; 614
гибкость объектов, 324
изменяемый, 675; 706
итерируемый, 152; 615; 624; 626
представления, 462
разделяемый, 212
сериализация объектов, 318
реза, 234
файла, 152
циклический, 337
Оператор, 344
=:, 497
break, 412; 414
continue, 412; 414
def, 496; 497
for, 417; 576
from, 704; 708; 724; 726
global, 514; 528
if, 394; 404
if/else, 405
import, 703; 708; 726
import spam, 734
nonlocal, 527; 528
pass, 412
print, 418
reload, 715
return, 494; 496
try, 358
while, 410; 576
выражения, 379
ограничители операторов, 401
присваивания, 362; 363
составной, 347; 394
Операторы Python
синтаксис, 345
Операции
над отображениями, 146
над последовательностями, 141; 306
Операция, 221
/ (деление), 180; 182
// (деление с округлением в меньшую
сторону), 179; 182
==, 328

- > < (надмножество, подмножество), 196
| (объединение), 196
& (пересечение), 196
- (разность), 196
^ (симметрическая разность, исключающее ИЛИ), 196
is, 328
абсолютного импортирования, 740
базовая, 230
 списковая, 271
булевская
 and, 408
 or, 407
вывода, 380
выражения, 196
импортирования внутри пакетов, 738
относительного импортирования, 736; 741
перегрузка операций, 173; 324
побитовая, 185
смешанные операции, 171
 старшинство операций, 171
Определение, 497
Оптимизация, 152
Отображения, 146
Очередь, 577
- П**
- Пакет, 720; 721
SciPy, 204
импортирование пакетов, 714; 720; 724
 относительное, 730
пакеты модулей, 730
подкаталоги пакета, 745
пространств имен, 748; 750
 произвольное вложение, 752
 семантика, 749
Параллелизм, 72
Переменная, 173; 208
 включения, 639
 локальная, 502; 609
Перестановки, 629
Подпрограмма (процедура), 493
Поиск
 конфигурирование пути поиска, 696
 настройка пути поиска, 721
Полиморфизм, 133; 161; 173; 217; 498; 501
Последовательность, 131; 221
 неизменяемая, 221
 обращение последовательности, 234
 операции над последовательностями, 306
 перемешивание последовательностей, 626
Поток
 автоматическое перенаправление потока, 387
- Правило LEGB, 507
Представления, 295
Привязка
 инициализации, 750
 объявление привязки, 741
Присваивание
 групповое, 371
 по индексу, 273
 по срезу, 273
Программа, 687
 архитектура программы Python, 687
Программирование
 системное, 48
 для баз данных, 50
 научное, 51
 объектно-ориентированное, 161
 функциональное, 54; 591
 численное, 51
Пространство имен, 105; 504
 автономное, 106
 модулей, 709
Протокол
 итерации, 442; 609; 614
 Python, 410
Прототипирование
 быстрое, 51
Псевдокод, 58
 исполняемый, 58
Путь
 настройка пути поиска, 721
 префиксы путей, 721
- Р**
- Расширение
 NumPy, 51
 ScientificPython, 51
 SciPy, 51
Рекурсия, 576; 630
 дополнительные примеры рекурсии, 579
- С**
- Сборка мусора, 148; 210
Связывание, 504
Система
 CPython, 70
 Cython, 73
 IronPython, 71
 Jython, 71
 Psyco, 74
 PyDoc, 466
 PyPy, 72
 Shed Skin, 74

Stackless, 72
Словарь, 129; 145; 146; 250; 268; 279; 300; 623
Сортировка, 275
Список, 129; 134; 141; 268; 307
изменение, 273
индексация, 272
итерация по спискам, 271
нарезание, 272
сортировка списков, 275
Сравнение
рекурсивное, 328
Среда
Eclipse, 115
Komodo, 115
NetBeans, 115
PythonWin, 115
разработки
интегрированная (IDLE), 108
Срез, 234; 428
Ссылка, 208
опережающая, 258
разделяемые ссылки, 212; 215
“слабая”, 217
счетчик ссылок, 209; 211
циклическая, 211
Стек, 577
LIFO, 277
Страйд, 233
Строка, 129; 131; 220; 223
Unicode, 219
байтовая, 140
документации, 469
изменение строк, 238; 241
инструменты преобразования строк, 235
строковые методы, 239; 240
текстовая, 140
форматирование строк, 176; 185; 246; 252
Строковый литерал, 222
Структура данных
разреженная, 288
Сценарий, 49; 89
инструмент написания сценариев, 54
написание сценариев для Интернета, 49
Сцепление
виртуальное, 751
Счетчик ссылок, 209; 211

T

Тестирование, 86
Тип, 142
булевский, 203
bool, 332

встроенный, 335
иерархии типов Python, 333
изменяемый, 214; 266
категории типов, 264
неизменяемый, 265
обозначение типа, 209
с плавающей точкой, 166
строковый, 220
str, 220
целое число, 166
числовой, 204
Типизация
динамическая, 56

У

Управляющие последовательности, 223

Ф

Файл, 154; 302; 309
py2app, 75
py2exe, 75
двоичный, 140; 156; 321
диспетчеры контекстов для файлов, 321
оптимизированного байт-кода, 699
с двоичными байтами, 155
с текстом Unicode, 156
текстовый, 140; 156
файловый итератор, 313; 623
фиксированный двоичный, 75
хранение объектов Python в файлах, 315

Формат
JSON, 318
числового отображения, 175

Форматирование, 135
строк, 185; 252

Функция, 492
apply, 556
__call__, 534
changer, 543
dir, 467; 469
enumerate, 433
eval, 184; 317
exec, 360
filter, 600; 618
format, 256
gensquares, 610
help, 472
input, 100
int, 235
intersect, 501
iter, 625
list, 242

map, 431; 594; 601; 607; 634; 653
min, 560
oct, 184
print, 100; 381; 383
range, 271; 425; 458; 603
reduce, 594
repr, 235
return, 608
str, 235
tester, 628
times, 499
yield, 608
zip, 431
анонимная, 585
атрибуты функций, 534; 582
вложенная, 519
встроенная, 187
генераторная, 462; 611; 619; 627
для работы с множествами, 563
доступа, 517
затруднения, связанные с функциями, 673
интроспекция функций, 581
операторы и выражения, связанные
 с функциями, 492
применение функций, 555
простая, 627
расширенные возможности функций, 571
рекурсивная, 573
фабричная, 520; 583
экранирование функций, 678

Ц

Цикл, 578
for, 371; 417; 426; 452
range, 425
while, 410
бесконечный, 411
вложенный, 452
интерактивный, 354
операторы цикла, 576

Ч

Число
десятичное, 130; 190
комплексное, 130; 183
рациональное, 130
с плавающей точкой, 130; 178
форматы числового отображения, 175
целое, 130

Я

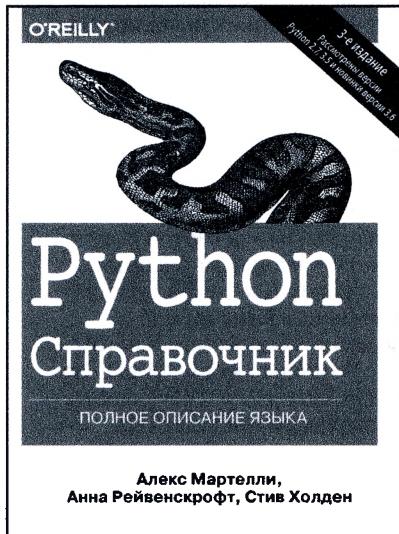
Язык
Python, 43
динамически типизированный, 129
объектно-ориентированный, 54
строго типизированный, 129

PYTHON СПРАВОЧНИК

ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА

3-Е ИЗДАНИЕ

**Алекс Мартелли
Анна Рейвенскрофт
Стив Холден**



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.

www.williamspublishing.com

ISBN 978-5-6040723-8-7

в продаже

СТАНДАРТНАЯ БИБЛИОТЕКА PYTHON 3

СПРАВОЧНИК С ПРИМЕРАМИ

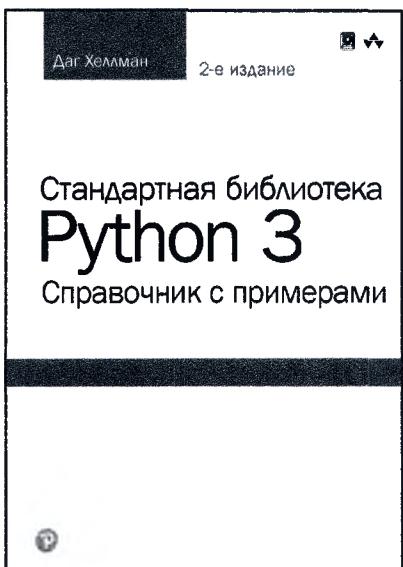
2-Е ИЗДАНИЕ

Даг Хеллман

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.



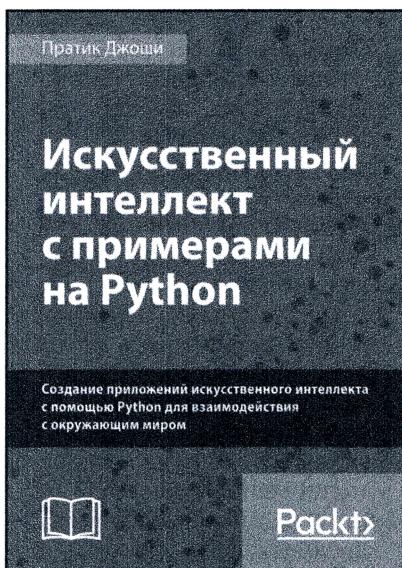
www.dialektika.com

ISBN: 978-5-6040043-8-8

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.

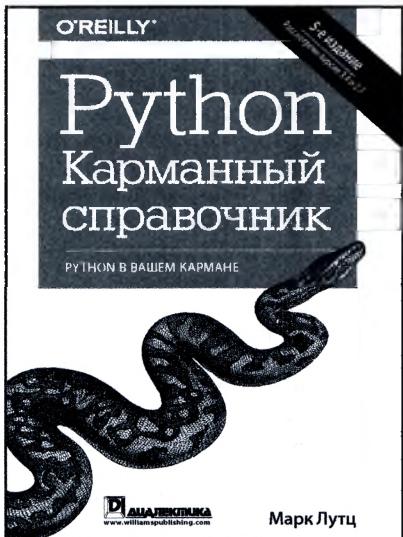
ISBN: 978-5-907114-41-8 в продаже

PYTHON

КАРМАННЫЙ СПРАВОЧНИК

ПЯТОЕ ИЗДАНИЕ

Марк Лутц



www.dialektika.com

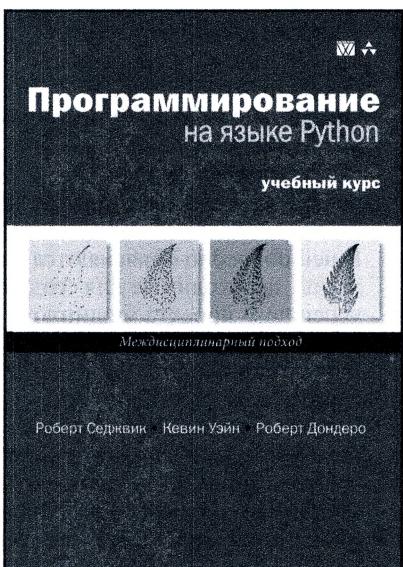
Этот краткий справочник по Python составлен с учетом версий 3.4 и 2.7 и очень удобен для наведения быстрых справок при разработке программ на Python. В лаконичной форме здесь представлены все необходимые сведения о типах данных и операторах Python, специальных методах перегрузки операторов, встроенных 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

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON

учебный курс

Роберт Седжвик
Кевин Уэйн
Роберт Дондеро



www.dialektika.com

Авторы книги сосредоточиваются на самых полезных и важных средствах языка Python и не стремятся к его абсолютноному охвату. Весь код этой книги был отработан и проверен на совместимость как с языком Python 2, так и Python 3, что делает его подходящим для каждого программиста и любого курса на многое лет вперед.

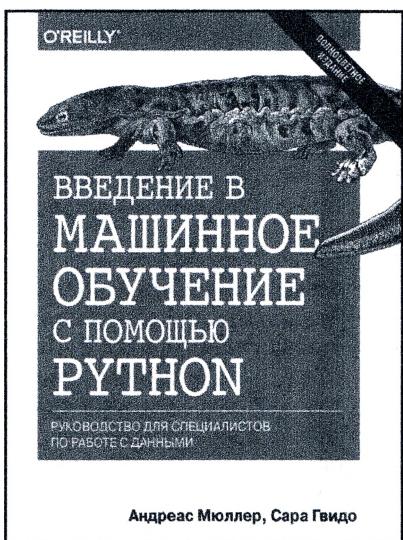
Особенности книги:

- всеобъемлющий, основанный на приложениях подход: изучение языка Python на примерах из области науки, математики, техники и коммерческой деятельности;
- основное внимание главному: самым полезным и важным средствам языка Python;
- совместимость примеров кода проверена на языках Python 2.x и Python 3.x;
- во все главы включены разделы с вопросами и ответами, упражнениями и практическими упражнениями.

ISBN 978-5-9908462-1-0 в продаже

ВВЕДЕНИЕ В МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ PYTHON РУКОВОДСТВО ДЛЯ СПЕЦИАЛИСТОВ ПО РАБОТЕ С ДАННЫМИ

**Андреас Мюллер
Сара Гвидо**



www.williamspublishing.com

ISBN 978-5-9908910-8-1

Эта полноцветная книга — отличный источник информации для каждого, кто собирается использовать машинное обучение на практике. Ныне машинное обучение стало неотъемлемой частью различных коммерческих и исследовательских проектов, и не следует думать, что эта область — прерогатива исключительно крупных компаний с мощными командами аналитиков. Эта книга научит вас практическим способам построения систем МО, даже если вы еще новичок в этой области. В ней подробно объясняются все этапы, необходимые для создания успешного проекта машинного обучения, с использованием языка Python и библиотек scikit-learn, NumPy и matplotlib. Авторы сосредоточили свое внимание исключительно на практических аспектах применения алгоритмов машинного обучения, оставив за рамками книги их математическое обоснование. Данная книга адресована специалистам, решающим реальные задачи, а поскольку область применения методов МО практически безгранична, прочитав эту книгу, вы сможете собственными силами построить действующую систему машинного обучения в любой научной или коммерческой сфере.

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**



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

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 978-5-8459-2090-4

в продаже