

Python

Лучшие практики и инструменты

ТРЕТЬЕ ИЗДАНИЕ

Михал Яворски, Тарек Зиаде



ББК 32.973.2-018.1
УДК 004.43
Я22

Яворски Михал, Зиаде Тарек

Я22 Python. Лучшие практики и инструменты. — СПб.: Питер, 2021. — 560 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1589-1

Python — это динамический язык программирования, используемый в самых разных предметных областях. Хотя писать код на Python просто, гораздо сложнее сделать этот код удобочитаемым, пригодным для многократного использования и легким в поддержке. Третье издание «Python. Лучшие практики и инструменты» даст вам инструменты для эффективного решения любой задачи разработки и сопровождения софта.

Авторы начинают с рассказа о новых возможностях Python 3.7 и продвинутых аспектах синтаксиса Python. Продолжают советами по реализации популярных парадигм, в том числе объектно-ориентированного, функционального и событийно-ориентированного программирования. Также авторы рассказывают о наилучших практиках именования, о том, какими способами можно автоматизировать развертывание программ на удаленных серверах. Вы узнаете, как создавать полезные расширения для Python на C, C++, Cython и CFFI.

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

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с Packt Publishing. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1789808896 англ.

© Packt Publishing 2019.

First published in the English language under the title 'Expert Python Programming – Third Edition – (9781789808896)'

ISBN 978-5-4461-1589-1

© Перевод на русский язык ООО Издательство «Питер», 2021

© Издание на русском языке, оформление ООО Издательство «Питер», 2021

© Серия «Библиотека программиста», 2021

© Перевод с англ. А. Павлов

Оглавление

О создателях книги	14
Об авторах.....	14
О научном редакторе.....	14
Предисловие	15
Для кого эта книга.....	15
Что мы рассмотрим	16
Как получить максимум от этой книги.....	17
Скачивание файлов с примерами кода.....	18
Скачивание цветных изображений.....	18
Условные обозначения	18
От издательства	20

Часть I. Перед началом работы

Глава 1. Текущее состояние Python	22
Технические требования.....	23
Где мы находимся и куда движемся	23
Почему и как изменился язык Python	23
Как не отставать от изменений в документации PEP	24
Внедрение Python 3 на момент написания этой книги	25
Основные различия между Python 3 и Python 2.....	26
Почему это должно нас волновать	26
Основные синтаксические различия и распространенные ошибки.....	27
Популярные инструменты и методы поддержания кросс-версионной совместимости	29
Не только CPython	33
Почему это должно нас волновать	33
Stackless Python	33
Jython	34
IronPython.....	35
PyPy	36
MicroPython	36
Полезные ресурсы	37
Резюме	38
Глава 2. Современные среды разработки на Python.....	39
Технические требования.....	40
Установка дополнительных пакетов Python с использованием pip.....	40
Изоляция сред выполнения.....	42
venv — виртуальное окружение Python	43
Изоляция среды на уровне системы	46
Виртуальные среды разработки, использующие Vagrant.....	47
Виртуальные среды, использующие Docker	49

Популярные инструменты повышения производительности	59
Пользовательские оболочки Python — ipython, bpython, ptpython и т. д.	60
Включение оболочек в собственные скрипты и программы	62
Интерактивные отладчики	63
Резюме	64

Часть II. Ремесло Python

Глава 3. Современные элементы синтаксиса — ниже уровня класса	66
Технические требования	67
Встроенные типы языка Python	67
Строки и байты	67
Контейнеры	73
Дополнительные типы данных и контейнеры	85
Специализированные контейнеры данных из модуля collections	85
Символическое перечисление с модулем enum	86
Расширенный синтаксис	88
Итераторы	88
Генераторы и операторы yield	91
Декораторы	94
Менеджеры контекста и оператор with	105
Функционально-стилевые особенности Python	109
Что такое функциональное программирование	110
Лямбда-функции	111
map(), filter() и reduce()	112
Частичные объекты и функция partial()	115
Выражения генераторов	116
Аннотации функций и переменных	117
Общий синтаксис	117
Возможные способы применения	118
Статическая проверка типа с помощью туру	118
Иные элементы синтаксиса, о которых вы, возможно, не знаете	119
Оператор for... else...	119
Именованные аргументы	120
Резюме	122
Глава 4. Современные элементы синтаксиса — выше уровня класса	123
Технические требования	124
Протоколы в языке Python — методы и атрибуты с двойным подчеркиванием	124
Сокращение шаблонного кода с помощью классов данных	126
Создание подклассов встроенных типов	128
ПРМ и доступ к методам из суперклассов	131
Классы старого стиля и суперклассы в Python 2	133
Понимание ПРМ в Python	134
Ловушки суперкласса	138
Практические рекомендации	141
Паттерны доступа к расширенным атрибутам	141
Дескрипторы	142
Свойства	147
Слоты	150
Резюме	151

Глава 5. Элементы метапрограммирования	152
Технические требования	152
Что такое метапрограммирование	153
Декораторы как средство метапрограммирования	153
Декораторы класса	154
Использование <code>__new__()</code> для переопределения процесса создания экземпляра	156
Метаклассы	158
Генерация кода	165
Резюме	172
Глава 6. Как выбирать имена	173
Технические требования	174
PEP 8 и практические рекомендации по именованию	174
Почему и когда надо соблюдать PEP 8	174
За пределами PEP 8 — правила стиля внутри команды	175
Стили именования	175
Переменные	176
Руководство по именованию	184
Использование префиксов <code>is/has</code> в булевых элементах	184
Использование множественного числа в именах коллекций	185
Использование явных имен для словарей	185
Избегайте встроенных и избыточных имен	185
Избегайте уже существующих имен	186
Практические рекомендации по работе с аргументами	187
Сборка аргументов по итеративному принципу	187
Доверие к аргументам и тестам	188
Осторожность при работе с магическими аргументами <code>*args</code> и <code>**kwargs</code>	188
Имена классов	190
Имена модулей и пакетов	191
Полезные инструменты	191
Pylint	192
pycodestyle и flake8	193
Резюме	194
Глава 7. Создаем пакеты	195
Технические требования	195
Создание пакета	196
Странности в нынешних инструментах создания пакетов в Python	196
Конфигурация проекта	198
Пользовательская команда <code>setup</code>	207
Работа с пакетами в процессе разработки	208
Пакеты пространства имен	209
Почему это полезно	210
Загрузка пакета	214
PyPI — каталог пакетов Python	214
Пакеты с исходным кодом и пакеты сборок	216
Исполняемые файлы	220
Когда бывают полезны исполняемые файлы	221
Популярные инструменты	221
Безопасность кода Python в исполняемых пакетах	228
Резюме	230

Глава 8. Развертывание кода	231
Технические требования.....	232
Двенадцатифакторное приложение	232
Различные подходы к автоматизации развертывания	234
Использование Fabric для автоматизации развертывания.....	235
Ваш собственный каталог пакетов или зеркало каталогов	239
Зеркала PyPI	240
Объединение дополнительных ресурсов с пакетом Python	241
Общие соглашения и практики	249
Иерархия файловой системы	249
Изоляция	250
Использование инструментов мониторинга процессов	250
Запуск кода приложения в пространстве пользователя.....	252
Использование обратного HTTP-прокси.....	253
Корректная перезагрузка процессов	254
Контрольно-проверочный код и мониторинг	256
Ошибки журнала — Sentry/Raven	256
Метрики систем мониторинга и приложений	260
Работа с журнальными приложениями.....	262
Резюме	267
Глава 9. Расширения Python на других языках	268
Технические требования.....	269
Различия между языками C и C++	269
Необходимость в использовании расширений.....	272
Повышение производительности критических фрагментов кода	272
Интеграция существующего кода, написанного на разных языках.....	273
Интеграция сторонних динамических библиотек.....	274
Создание пользовательских типов данных	274
Написание расширений.....	275
Расширения на чистом языке C	276
Написание расширений на Cython	291
Проблемы с использованием расширений	295
Дополнительная сложность.....	296
Отладка	297
Взаимодействие с динамическими библиотеками без расширений	297
Модуль ctypes	298
CFFI	304
Резюме	306

Часть III. Качество, а не количество

Глава 10. Управление кодом	308
Технические требования.....	308
Работа с системой управления версиями	308
Централизованные системы	309
Распределенные системы.....	312
Распределенные стратегии	313
Централизованность или распределенность.....	314
По возможности используйте Git.....	315
Рабочий процесс GitFlow и GitHub Flow	316

Настройка процесса непрерывной разработки	320
Непрерывная интеграция	321
Непрерывная доставка	325
Непрерывное развертывание	326
Популярные инструменты для непрерывной интеграции	326
Выбор правильного инструмента и распространенные ошибки	335
Резюме	338
Глава 11. Документирование проекта	339
Технические требования	339
Семь правил технической документации	340
Пишите в два этапа	340
Ориентируйтесь на читателя	341
Упрощайте стиль	342
Ограничивайте объем информации	342
Используйте реалистичные примеры кода	343
Пишите по минимуму, но достаточно	344
Используйте шаблоны	344
Документация как код	345
Использование строк документации в Python	345
Популярные языки разметки и стилей для документации	347
Популярные генераторы документации для библиотек Python	348
Sphinx	349
MkDocs	352
Сборка документации и непрерывная интеграция	352
Документирование веб-API	353
Документация как прототип API с API Blueprint	354
Самодокументирующиеся API со Swagger/OpenAPI	355
Создание хорошо организованной системы документации	356
Создание портфеля документации	356
Ваш собственный портфель документации	362
Создание шаблона документации	363
Шаблон для автора	364
Шаблон для читателя	364
Резюме	365
Глава 12. Разработка на основе тестирования	366
Технические требования	366
Я не тестирую	367
Три простых шага разработки на основе тестирования	367
О каких тестах речь	372
Стандартные инструменты тестирования в Python	375
Я тестирую	380
Ловушки модуля unittest	380
Альтернативы модулю unittest	381
Охват тестирования	388
Подделки и болванки	390
Совместимость среды тестирования и зависимостей	396
Разработка на основе документации	400
Резюме	402

Часть IV. Жажда скорости

Глава 13. Оптимизация — принципы и методы профилирования	404
Технические требования	404
Три правила оптимизации	405
Сначала — функционал	405
Работа с точки зрения пользователя	406
Поддержание читабельности и удобства сопровождения	407
Стратегии оптимизации	408
Пробуем свалить вину на другого	408
Масштабирование оборудования	409
Написание теста скорости	410
Поиск узких мест	410
Профилирование использования ЦП	411
Профилирование использования памяти	419
Профилирование использования сети	430
Резюме	433
Глава 14. Эффективные методы оптимизации	434
Технические требования	435
Определение сложности	436
Цикломатическая сложность	437
Нотация «О большое»	438
Уменьшение сложности через выбор подходящей структуры данных	440
Поиск в списке	440
Использование модуля collections	442
Тип deque	442
Тип defaultdict	444
Тип namedtuple	444
Использование архитектурных компромиссов	446
Использование эвристических алгоритмов или приближенных вычислений	446
Применение очереди задач и отложенная обработка	447
Использование вероятностной структуры данных	450
Кэширование	451
Детерминированное кэширование	452
Недетерминированное кэширование	455
Сервисы кэширования	456
Резюме	460
Глава 15. Многозадачность	461
Технические требования	461
Зачем нужна многозадачность	462
Многопоточность	463
Что такое многопоточность	464
Как Python работает с потоками	465
Когда использовать многопоточность	466
Многопроцессорная обработка	481
Встроенный модуль multiprocessing	483
Асинхронное программирование	489
Кооперативная многозадачность и асинхронный ввод/вывод	490
Ключевые слова async и await	491

Модуль <code>asyncio</code> в старых версиях Python	495
Практический пример асинхронного программирования	495
Интеграция синхронного кода с помощью фьючерсов <code>asupc</code>	498
Резюме	501

Часть V. Техническая архитектура

Глава 16. Событийно-ориентированное и сигнальное программирование	504
Технические требования.....	505
Что такое событийно-ориентированное программирование	505
Событийно-ориентированный != асинхронный.....	506
Событийно-ориентированное программирование в GUI.....	507
Событийно-ориентированная связь	509
Различные стили событийно-ориентированного программирования.....	511
Стиль на основе обратных вызовов.....	511
Стиль на основе субъекта	513
Тематический стиль	515
Событийно-ориентированные архитектуры	518
Очереди событий и сообщений	519
Резюме	521
Глава 17. Полезные паттерны проектирования	523
Технические требования.....	524
Порождающие паттерны	524
Синглтон.....	524
Структурные паттерны.....	527
Адаптер	528
Заместитель.....	542
Фасад.....	543
Поведенческие паттерны	544
Наблюдатель	544
Посетитель	546
Шаблонный метод.....	548
Резюме	550
Приложение. <code>reStructuredText Primer</code>	552
<code>reStructuredText</code>	552
Структура раздела	554
Списки	555
Форматирование внутри строк	556
Блок литералов.....	557
Ссылки.....	558

3

Современные элементы синтаксиса — ниже уровня класса

Язык Python за последние несколько лет серьезно эволюционировал. С выхода самой ранней версии и до текущего момента (версия 3.7) было введено много усовершенствований, которые позволили сделать его более чистым и простым. Основы Python не изменились, но предоставляемые им инструменты стали гораздо более эргономичными.

По мере развития Python ваше ПО тоже должно эволюционировать. Если вы уделите достаточно внимания тому, как пишется программа, то это очень поможет ее эволюции. Многие программы в конечном итоге пришлось переписать с нуля из-за деревянного синтаксиса, неясного API или нетрадиционных стандартов. Использование новых возможностей языка программирования, которые позволяют сделать код более выразительным и читабельным, повышает сопровождаемость программного обеспечения и тем самым продлевает срок его службы.

В этой главе мы рассмотрим наиболее важные элементы современного синтаксиса Python, а также советы по их использованию. Мы также обсудим детали, связанные с реализацией встроенных типов Python, которые по-разному влияют на производительность кода, но при этом не станем чрезмерно углубляться в методы оптимизации. Советы по повышению производительности кода, ускорению работы или оптимизации использования памяти будут представлены позже, в главах 13 и 14.

В этой главе:

- ❑ встроенные типы языка Python;
- ❑ дополнительные типы данных и контейнеры;
- ❑ расширенный синтаксис;
- ❑ функционально-стилевые особенности Python;
- ❑ аннотации функций и переменных;
- ❑ другие элементы синтаксиса, о которых вы, возможно, не знаете.

Технические требования

Файлы с примерами кода для этой главы можно найти по ссылке github.com/packtpublishing/expert-python-programming-third-edition/tree/master/chapter3.

Встроенные типы языка Python

В Python предусмотрен большой набор типов данных, как числовых, так и типов-коллекций. В синтаксисе числовых типов нет ничего особенного. Конечно, существуют некоторые различия в определении литералов каждого типа и несколько не очень хорошо известных деталей касательно операторов, но в целом синтаксис числовых типов в Python мало чем может вас удивить. Однако все меняется, когда речь заходит о коллекциях и строках. Несмотря на правило *«каждому действию — один способ»*, разработчик на Python часто располагает немалым количеством вариантов. Некоторые шаблоны кода, кажущиеся новичкам интуитивно понятными и простыми, опытные программисты нередко считают «неканоническими», поскольку те либо неэффективны, либо слишком многословны.

«Пайтоноподобные» паттерны для решения часто встречающихся задач (многие программисты называют их идиомами) часто могут показаться лишь эстетическим украшением. Но это в корне неверно. Большинство идиом порождены тем, как Python реализован внутри и как работают его встроенные конструкции и модули. Зная больше о таких деталях, вы можете более глубоко и правильно понимать принципы работы языка. К сожалению, в сообществе существуют некие мифы и стереотипы о том, как работает Python. Только самостоятельно углубившись в изучение языка, вы сможете понять, что из них правда, а что — ложь.

Посмотрим на строки и байты.

Строки и байты

Тема строк может привести некоторую путаницу для программистов, которые раньше работали только в Python 2. В Python 3 существует лишь один тип данных, способный хранить текстовую информацию, — `str`, то есть просто строка. Это неизменяемая последовательность, хранящая кодовые точки Unicode. В этом состоит основное отличие от Python 2, где тип `str` представлял собой строки байтов, которые в настоящее время обрабатываются объектами `byte` (но не точно таким же образом).

Строки в Python являются последовательностями. Одного этого факта должно быть достаточно, чтобы включить их обсуждение в раздел, посвященный другим типам контейнеров. Но строки отличаются от других типов контейнеров одной важной деталью. Они имеют весьма специфические ограничения на тип данных, который могут хранить, — а именно, текст Unicode.

Тип `byte` (и его изменяемая альтернатива `bytearray`) отличается от `str` тем, что принимает только байты в качестве значения последовательности, а байты в Python являются целыми числами в диапазоне $0 \leq x < 256$. Поначалу это может показаться сложным, поскольку при выводе на печать байты могут быть очень похожи на строки:

```
>>> print(bytes([102, 111, 111]))
b'foo'
```

Типы `byte` и `bytearray` позволяют работать с сырыми двоичными данными, которые не всегда могут быть текстовыми (например, аудио- и видеофайлы, изображения и сетевые пакеты). Истинная природа этих типов вскрывается, когда они превращаются в другие типы последовательностей, такие как списки или кортежи:

```
>>> list(b'foo bar')
[102, 111, 111, 32, 98, 97, 114]
>>> tuple(b'foo bar')
(102, 111, 111, 32, 98, 97, 114)
```

В Python 3 велось немало споров о нарушении обратной совместимости для строковых литералов и о том, как язык обрабатывает Unicode. Начиная с Python 3.0, каждый строковый литерал без префикса обрабатывается как Unicode. Литералы, заключенные в одинарные кавычки (`'`), двойные кавычки (`"`) или группы из трех кавычек (одинарных или двойных) без префикса, представляют тип данных `str`:

```
>>> type("some string")
<class 'str'>
```

В Python 2 литералы Unicode требуют префикса (например, `u"строка"`). Этот префикс по-прежнему разрешен для сохранения обратной совместимости (начиная с Python 3.3), но не имеет никакого синтаксического значения в Python 3.

Байтовые литералы уже были представлены в некоторых предыдущих примерах, но их синтаксис будет показан для сохранения целостности повествования. Байтовые литералы заключены в одиночные, двойные или тройные кавычки, но им должен предшествовать префикс `b` или `B`:

```
>>> type(b"some bytes")
<class 'bytes'>
```

Обратите внимание: в Python нет синтаксиса для литералов `bytearray`. Если вы хотите создать значение `bytearray`, то вам нужно использовать литерал `bytes` и конструктор типа `bytearray()`:

```
>>> bytearray(b'some bytes')
bytearray(b'some bytes')
```

Важно помнить, что в строках Unicode содержится абстрактный текст, который не зависит от представления байтов. Это делает их непригодными для сохранения

на диске или отправки по сети без перекодирования в двоичные данные. Есть два способа кодирования строки объектов в последовательность байтов.

- ❑ С помощью метода `str.encode(encoding, errors)`, который кодирует строку, используя имеющийся кодировщик. Кодировщик задается через аргумент `encoding`, по умолчанию равный UTF-8. Второй аргумент задает схему обработки ошибок. Может принимать значения `'strict'` (по умолчанию), `'ignore'`, `'replace'`, `'xmlcharrefreplace'` или любой другой зарегистрированный обработчик (см. документацию модуля `codecs`).
- ❑ С помощью конструктора `bytes(source, encoding, errors)`, который создает новую последовательность байтов. Когда `source` имеет тип `str`, аргумент `encoding` является обязательным и не имеет значения по умолчанию. Аргументы `encoding` и `errors` такие же, как и для метода `str.encode()`.

Двоичные данные, представленные типом `bytes`, могут быть преобразованы в строку аналогичным образом.

- ❑ С помощью метода `bytes.decode(encoding, errors)`, который декодирует байты с использованием имеющегося кодировщика. Аргументы этого метода имеют тот же смысл и значение по умолчанию, что и у `str.encode()`.
- ❑ С помощью конструктора `str(source, encoding, errors)`, который создает новый экземпляр строки. Как и в конструкторе `bytes()`, кодирующий аргумент `encoding` является обязательным и не имеет значения по умолчанию, если в качестве `source` используется последовательность байтов.



Байты или строка байтов: путаница в именах

Из-за изменений, внесенных в Python 3, некоторые программисты считают экземпляры `bytes` байтовыми строками. В основном это связано с историческими причинами: `bytes` в Python 3 является типом, наиболее близким к типу `str` из Python 2 (но это не одно и то же). Тем не менее экземпляр `bytes` представляет собой последовательность байтов и не обязательно несет в себе текстовые данные. Чтобы избежать путаницы, рекомендуется всегда ссылаться на них как на байты или последовательность байтов, несмотря на их сходство со строками. Понятие строк в Python 3 зарезервировано для текстовых данных, и это всегда тип `str`.

Рассмотрим подробности реализации строк и байтов.

Детали реализации

Строки в Python являются неизменяемыми. Это верно и для байтовых последовательностей. Данный факт очень важен, поскольку имеет как преимущества, так и недостатки. Вдобавок он влияет и на то, как именно эффективно обрабатывать

строки в Python. Благодаря своей неизменности строки могут быть использованы в качестве ключей в словарях или в качестве элемента множества, поскольку после инициализации они не меняют значения. С другой стороны, всякий раз, когда вам требуется измененная строка (даже с крошечной модификацией), придется создавать новый экземпляр. К счастью, у `bytearray`, изменяемого варианта `bytes`, такой проблемы нет. Массивы байтов могут быть изменены «на месте» (без создания новых объектов) через присвоение элементов, а могут изменяться динамически, так же как списки — с помощью склеивания, вставок и т. д.

Поговорим подробнее о конкатенации.

Конкатенация строк

Неизменяемость строк в Python создает некоторые проблемы, когда нужно объединять несколько экземпляров строк. Ранее мы уже отмечали, что конкатенация неизменяемых последовательностей приводит к созданию нового объекта-последовательности. Представим, что новая строка строится путем многократной конкатенации нескольких строк, как показано ниже:

```
substrings = ["These ", "are ", "strings ", "to ", "concatenate."]
s = ""
for substring in substrings:
    s += substring
```

Это ведет к квадратичным затратам времени выполнения в зависимости от общей длины строки. Другими словами, крайне неэффективно. Для обработки таких ситуаций предусмотрен метод `str.join()`. Он принимает в качестве аргумента итерируемые величины или строки и возвращает объединенные строки. Вызов метода `join()` на строках можно выполнить двумя способами:

```
# Используем пустой литерал
s = "".join(substrings)

# Используем «неограниченный» вызов метода
str.join("", substrings)
```

Первая форма вызова `join()` является наиболее распространенной идиомой. Строка, которая вызывает этот метод, будет использоваться в качестве разделителя между подстроками. Рассмотрим следующий пример:

```
>>> ','.join(['some', 'comma', 'separated', 'values'])
'some,comma,separated,values'
```

Стоит помнить: преимущества по быстродействию (особенно для больших списков) недостаточно, чтобы метод `join()` стал панацеей в любой ситуации,

когда нужно объединить две строки. Несмотря на широкое признание, эта идиома не улучшает читабельность кода. А читабельность очень важна! Кроме того, бывают ситуации, когда метод `join()` не будет работать так же хорошо, как обычная конкатенация с оператором `+`. Вот несколько примеров.

- ❑ Если подстрока очень мало и они не содержатся в итерируемой переменной (существующий список или кортеж строк), то в некоторых случаях затраты на создание новой последовательности только для выполнения конкатенации могут свести на нет преимущество `join()`.
- ❑ При конкатенации коротких литералов благодаря некоторой оптимизации интерпретатора, например сворачиванию в константы в CPython (см. следующий подпункт), часть сложных литералов (не только строки), таких как `'a' + 'b' + 'c'`, может быть переведена в более короткую форму во время компиляции (здесь `'abc'`). Конечно, это разрешено только для относительно коротких констант (литералов).

В конечном счете если количество строк для конкатенации известно заранее, то лучшая читабельность обеспечивается надлежащим форматированием строки с помощью метода `str.format()`, оператора `%`, или форматирования f-строк. В разделах кода, где производительность не столь важна или выигрыш от оптимизации конкатенации очень мал, форматирование строк — лучшая альтернатива конкатенации.

Сворачивание, локальный оптимизатор и оптимизатор AST. В CPython существуют различные методы оптимизации кода. Первая оптимизация выполняется, как только исходный код преобразуется в форму абстрактного синтаксического дерева перед компиляцией в байт-код. CPython может распознавать определенные закономерности в абстрактном синтаксическом дереве и вносить в него прямые изменения. Другой вид оптимизации — локальная. В ней реализуется ряд общих оптимизаций непосредственно в байт-коде Python. Мы уже упоминали ранее, что сворачивание в константы — одно из таких свойств. Оно позволяет интерпретатору преобразовывать сложные буквенные выражения (такие как `"one" + " " + "thing"`, `" * 79` или `60 * 1000`) в один литерал, который не требует дополнительных операций (конкатенации или умножения) во время выполнения.

До Python 3.5 все сворачивание в константы выполнялось в CPython только локальным оптимизатором. В случае со строками полученные константы были ограничены по длине с помощью закодированного значения. В Python 3.5 это значение было равно 20. В Python 3.7 большинство оптимизаций сворачивания обрабатывается на уровне абстрактного синтаксического дерева. Но это скорее забавные факты, а не полезные сведения. Информацию о других интересных оптимизациях,

выполняемых AST и локальным оптимизатором, можно найти в файлах исходного кода `Python/ast_opt.c` и `Python/peephole.c`.

Рассмотрим форматирование f-строками.

Форматирование f-строками

F-строки — одна из самых любимых новых функций Python, которая появилась в Python 3.6. Это также одна из самых противоречивых особенностей данной версии. F-строки, или *форматированные строковые литералы*, введенные в документе PEP 498, — новый инструмент форматирования строк в Python. До Python 3.6 существовало два основных способа форматирования строк:

- ❑ с помощью `%`, например `"Some string with included % value" % "other";`
- ❑ с помощью метода `str.format()`, например `"Some string with included {other} value".format(other="other")`.

Форматированные строковые литералы обозначаются префиксом `f`, и их синтаксис наиболее близок к методу `str.format()`, поскольку они используют подобную разметку для обозначения замены полей в тексте, который должен быть отформатирован. В методе `str.format()` замена текста относится к аргументам и именованным аргументам, передаваемым в метод форматирования. Вы можете использовать как анонимные замены, которые будут превращаться в последовательные индексы аргументов, так и явные индексы аргументов или имена ключевых слов.

Это значит, что одна и та же строка может быть отформатирована по-разному:

```
>>> from sys import version_info
>>> "This is Python {}.{}".format(*version_info)
'This is Python 3.7'

>>> "This is Python {0}.{1}".format(*version_info)
'This is Python 3.7'

>>> "This is Python {major}.{minor}".format(major=version_info.major,
minor=version_info.minor)
'This is Python 3.7'
```

Особенности f-строки делает тот факт, что заменяемые поля могут быть любым выражением Python, которое вычисляется во время выполнения. Внутри строк у вас есть доступ к любой переменной, доступной в том же пространстве имен, что и форматированный литерал. С помощью f-строк предшествующие примеры можно записать следующим образом:

```
>>> from sys import version_info
>>> f"This is Python {version_info.major}.{version_info.minor}"
'This is Python 3.7'
```

Возможность использовать выражения в заменяемых полях позволяет упростить форматирование кода. Можно применять те же спецификаторы форматирования (заполнение пробелов, выравнивание, разметку и т. д.), как и в методе `str.format()`. Синтаксис выглядит следующим образом:

```
f"{replacement_field_expression:format_specifier}"
```

Ниже приведен простой пример кода, который печатает первые десять степеней числа 10, используя f-строку, и выравнивает результаты, используя строковое форматирование с заполнением пробелами:

```
>>> for x in range(10):
...     print(f"10^{x} == {10**x:10d}")
...
10^0 ==      1
10^1 ==     10
10^2 ==    100
10^3 ==   1000
10^4 ==  10000
10^5 == 100000
10^6 == 1000000
10^7 == 10000000
10^8 == 100000000
10^9 == 1000000000
```

Полная спецификация форматирования строк в Python — это почти еще один язык программирования внутри Python. Лучшим справочником по форматированию будет официальная документация, которую можно найти по адресу docs.python.org/3/library/string.html. Еще один полезный интернет-ресурс по данной теме: pyformat.info. Он содержит наиболее важные элементы этой спецификации, сопровождаемые примерами.

В следующем подразделе мы рассмотрим коллекции языка.

Контейнеры

В Python предусмотрен неплохой выбор встроенных контейнеров данных, позволяющих эффективно решать многие проблемы, если подойти к выбору с умом. Типы, которые вы уже должны знать, имеют специальные литералы:

- ❑ списки;
- ❑ кортежи;
- ❑ словари;
- ❑ множества.

Разумеется, Python не ограничивается этими четырьмя контейнерами. Ассортимент можно серьезно расширить с помощью стандартной библиотеки. Часто решения

некоторых проблем сводятся к правильному выбору структуры данных для их хранения. Эта часть книги призвана облегчить принятие подобных решений и помочь лучше понять возможные варианты.

Списки и кортежи

Два основных типа коллекций в Python — списки и кортежи, и оба они представляют собой последовательность объектов. Основное различие между ними должно быть очевидно для любого, кто изучает Python чуть больше пары часов: списки являются динамическими и их размер может изменяться, в то время как кортежи неизменяемы.

Списки и кортежи в Python претерпели немало оптимизаций, которые позволяют ускорить выделение/очистку памяти для небольших объектов. Кроме того, строки и кортежи рекомендуются для типов данных структур, где позиция элемента — информация, полезная сама по себе. Например, кортежи отлично подходят для хранения пар координат (x, y) . Детали реализации кортежей интереса не представляют. Важно в рамках данной главы только то, что `tuple` является *неизменяемым* и, следовательно, *хешируемым*. Подробное объяснение будет приведено в подразделе, посвященном словарям. Динамический аналог кортежей, а именно списки, для нас интереснее. Ниже мы обсудим их функционирование и то, как эффективно работать с ними.

Детали реализации. Многие программисты часто путают тип `list` Python со связанными списками, которые обычно встречаются в стандартных библиотеках других языков, таких как C, C++ или Java. На самом деле списки CPython — вообще не списки. В CPython списки реализованы в виде массивов переменной длины. Это работает и для других реализаций, таких как Jython и IronPython, хотя подобные детали не всегда бывают задокументированы. Причины такой путаницы понятны: этот тип данных называется *списком* и имеет интерфейс, типичный для любой имплементации структуры данных «связный список».

Почему это важно и что это значит? Списки — одна из наиболее популярных структур данных, и то, как они используются, в значительной степени влияет на производительность приложения. CPython — наиболее популярная и используемая реализация, поэтому невероятно важно знать, как она устроена.

Списки в Python представляют собой непрерывные массивы ссылок на другие объекты. Указатель на данный массив и значение длины хранятся в головной структуре списка. Это значит, что каждый раз, когда в список добавляется или из списка удаляется элемент, массив ссылок переопределяется (с точки зрения памяти). К счастью, в Python эти массивы создаются с экспоненциальным избыточным выделением, вследствие чего не каждая операция требует фактического изменения размера базового массива. Поэтому затраты на выполнение мелких изменений на самом деле не столь велики. К сожалению, другие операции, ко-

которые считаются *быстрыми* в обычных связанных списках, в Python имеют относительно высокую вычислительную сложность:

- ❑ вставка элемента в произвольном месте с использованием метода `list.insert` имеет сложность $O(n)$;
- ❑ удаление элемента с помощью `list.delete` или с помощью оператора `del` имеет сложность $O(n)$.

Извлечение или установка элемента по индексу — это операция, сложность которой не зависит от размера списка и всегда равна $O(1)$.

Пусть n — длина списка. Вычислительная сложность для большинства операций со списками приведена в табл. 3.1.

Таблица 3.1

Операция	Сложность
Копия	$O(n)$
Присоединение	$O(1)$
Вставка	$O(n)$
Извлечение значения элемента	$O(1)$
Установка значения элемента	$O(1)$
Удаление элемента	$O(n)$
Итерация	$O(n)$
Извлечение среза длины k	$O(k)$
Удаление среза	$O(n)$
Установка среза длины k	$O(k+n)$
Расширение	$O(n)$
Умножение на k	$O(nk)$
Проверка существования (элемента в списке)	$O(n)$
<code>min()/max()</code>	$O(n)$
Возврат длины	$O(1)$

Если необходим реальный связанный или дважды связанный список, то в Python есть тип `deque` во встроенном модуле `collections`. Эта структура данных позволяет добавлять и удалять элементы с каждой стороны со сложностью $O(1)$. Это обобщение стеков и очередей, которое должно нормально работать в задачах, где требуется дважды связанный список.

Списковое включение. Как вы, наверное, знаете, написание подобного кода может быть утомительным:

```
>>> evens = []
>>> for i in range(10):
...     if i % 2 == 0:
...         evens.append(i)
...
>>> evens
[0, 2, 4, 6, 8]
```

Это может работать на C, однако на Python замедляет работу по следующим причинам:

- ❑ заставляет интерпретатор работать каждый цикл, чтобы определить, какую часть последовательности нужно изменить;
- ❑ заставляет вводить отдельный счетчик, который отслеживает, какой элемент обрабатывается;
- ❑ нужно выполнять дополнительный просмотр на каждой итерации, поскольку `append()` является методом списков.

Списковое включение лучше всего подходит для такого рода ситуаций. Оно позволяет определить список с помощью одной строки кода:

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

Запись такого вида намного короче и включает в себя меньше элементов. Для большой программы это значит меньше ошибок и код, который легче читать. Именно поэтому многие опытные программисты на Python будут считать такие формы более удобочитаемыми.



Списковое включение и изменение размера массива

Среди некоторых Python-программистов бытует такой миф: списковое включение позволяет обойти тот факт, что внутренний массив, представляющий объект списка, меняет свой размер после каждого изменения. Некоторые говорят, что памяти для массива выделяется ровно столько, сколько нужно. К сожалению, это не так.

Интерпретатор, оценивая включение, не может знать, насколько велик будет окончательный контейнер, и не может заранее выделить нужный объем памяти. Поэтому внутренний массив определяется по той же схеме, которая была бы при использовании цикла `for`. Тем не менее во многих случаях создать список с помощью включения будет чище и быстрее, чем с применением обычных циклов.

Другие идиомы. Другой типичный пример идиом Python — использование встроенной функции `enumerate()`. Она предоставляет удобный способ получить индекс, когда последовательность итерируется внутри цикла. Рассмотрим следующий фрагмент кода в качестве примера отслеживания индекса элемента без функции `enumerate()`:

```
>>> i = 0
>>> for element in ['one', 'two', 'three']:
...     print(i, element)
...     i += 1
...
0 one
1 two
2 three
```

Этот фрагмент можно заменить следующим кодом, который будет короче и, безусловно, чище:

```
>>> for i, element in enumerate(['one', 'two', 'three']):
...     print(i, element)
...
0 one
1 two
2 three
```

Если необходимо объединить элементы нескольких списков (или любых других итерируемых типов) «один за одним», то можно использовать встроенную функцию `zip()`. Ниже приведен стандартный код для равномерного прохода по двум итерируемым объектам одного размера:

```
>>> for items in zip([1, 2, 3], [4, 5, 6]):
...     print(items)
...
(1, 4)
(2, 5)
(3, 6)
```

Обратите внимание, что результаты функции `zip()` можно отменить путем вызова другой функции `zip()`:

```
>>> for items in zip(*zip([1, 2, 3], [4, 5, 6])):
...     print(items)
...
(1, 2, 3)
(4, 5, 6)
```

О функции `zip()` важно помнить следующее: она ожидает, что вводимые итерируемые объекты будут одинакового размера. Если вы введете аргументы разной

длины, то вывод будет сформирован для короткого аргумента, как показано в следующем примере:

```
>>> for items in zip([1, 2, 3, 4], [1, 2]):
...     print(items)
...
(1, 1)
(2, 2)
```

Еще один популярный элемент синтаксиса — последовательная распаковка. Она не ограничивается списками и кортежами и будет работать с любым типом последовательности (даже со строками и последовательностями байтов). Она позволяет распаковывать последовательность элементов в другой набор переменных, до тех пор пока с левой стороны от оператора присваивания есть столько же переменных, сколько элементов в последовательности. Если вы внимательно читали фрагменты кода, то, возможно, уже отметили эту идиому, когда мы обсуждали функцию `enumerate()`.

Ниже приведен специальный пример этого синтаксического элемента:

```
>>> first, second, third = "foo", "bar", 100
>>> first
'foo'
>>> second
'bar'
>>> third
100
```

Кроме того, распаковка позволяет хранить несколько элементов в одной переменной с помощью выражений со звездочкой, если такое выражение может быть однозначно истолковано. Распаковка также может выполняться с вложенными последовательностями. Это может быть полезно, особенно при переборе некоторых сложных структур данных, составленных из нескольких последовательностей. Ниже приведены примеры более сложной распаковки последовательностей:

```
>>> # Захват конца последовательности
>>> first, second, *rest = 0, 1, 2, 3
>>> first
0
>>> second
1
>>> rest
[2, 3]
>>> # Захват середины последовательности
>>> first, *inner, last = 0, 1, 2, 3
>>> first
0
>>> inner
```



```
[1, 2]
>>> last
3
>>> # Распаковка иерархии
>>> (a, b), (c, d) = (1, 2), (3, 4)
>>> a, b, c, d
(1, 2, 3, 4)
```

Словари

Словари — одна из наиболее универсальных структур данных в Python. Тип `dict` позволяет сопоставить набор уникальных ключей со значениями следующим образом:

```
{
    1: ' one',
    2: ' two',
    3: ' three',
}
```

По идее, вы уже должны знать словарные литералы — в них нет ничего сложного. Python позволяет программистам также создать новый словарь, используя выражения генерации списков. Ниже приведен простой пример кода, который возводит числа в диапазоне от 0 до 99 в их квадраты:

```
squares = {number: number**2 for number in range(100)}
```

Важно то, что вся мощь генерации списков доступна и в словарях. Поэтому они часто бывают более эффективны и делают код короче и чище. Для более сложного кода, в котором для создания словаря требуется много операторов `if` или вызовов функций, подойдет простой цикл `for`, особенно если это улучшает читабельность.

Программистам, которым Python 3 в новинку, следует знать важную информацию об итерировании словарных элементов. Методы словарей `keys()`, `values()` и `items()` больше не возвращают списков. Кроме того, их аналоги, `iterkeys()`, `itervalues()` и `iteritems()`, возвращающие итераторы, в Python 3 вообще отсутствуют. Теперь методы `keys()`, `values()` и `items()` возвращают специальные объекты-представления:

- ❑ `keys()` — возвращает объект `dict_keys`, в котором перечислены все ключи словаря;
- ❑ `values()` — возвращает объект `dict_values`, в котором перечислены все значения словаря;
- ❑ `items()` — возвращает объект `dict_items`, в котором перечислены пары «ключ — значение» в виде кортежей.

Объект-представление позволяет просматривать контент словаря динамическим образом, и каждый раз, когда в словарь вносятся изменения, они появляются и в данном объекте:

```
>>> person = {'name': 'John', 'last_name': 'Doe'}
>>> items = person.items()
>>> person['age'] = 42
>>> items
dict_items([('name', 'John'), ('last_name', 'Doe'), ('age', 42)])
```

Объекты-представления ведут себя как в старые времена вели себя списки, возвращаемые методом `iter()`. Эти объекты не хранят все значения в памяти (как, например, списки), но позволяют узнать их длину (с помощью функции `len()`) и проверять наличие (ключевое слово `in`). И еще они, конечно, итерируемые.

Еще одна важная особенность объектов-представлений заключается в том, что результат методов `keys()` и `values()` дает одинаковый порядок ключей и значений. В Python 2 нельзя изменять содержимое словаря между вызовами этих методов, если вы хотите получить одинаковый порядок извлекаемых ключей и значений. Теперь объекты `dict_keys` и `dict_values` динамические, так что даже если содержание словаря изменяется между вызовами методов, то порядок итерации будет подстроен соответствующим образом.

Подробности реализации. В CPython в качестве базовой структуры данных для словарей используются хеш-таблицы с псевдослучайным зондированием. Это выглядит как излишние дебри реализации, но в ближайшем будущем здесь вряд ли что-то изменится, и это довольно интересный факт для программиста на Python.

Из-за данной особенности реализации в качестве ключей в словарях могут использоваться только *хешируемые* (hashable) объекты. Таковым является объект, имеющий значение хеш-функции, которое не меняется в течение срока его существования, и его можно сравнивать с другими объектами. Каждый встроенный неизменяемый тип Python — хешируемый. Изменяемые типы, такие как списки, словари и множества, не являются таковыми и поэтому не могут быть использованы в качестве ключей словаря. Протокол, определяющий хешируемость типа, состоит из двух методов:

- ❑ `__hash__` — возвращает хеш-значение (целочисленное), которое необходимо для внутренней реализации типа `dict`. Для объектов — экземпляров пользовательских классов является производным от `id()`;
- ❑ `__eq__` — проверяет два объекта на предмет одинаковости их значений. Все объекты, которые являются экземплярами пользовательских классов, по умолчанию не равны, если не сравниваются сами с собой.

Два проверяемых на равенство объекта должны иметь одинаковое значение хеш-функции. При этом обратное утверждение не обязательно верно. То есть возможны конфликты хеша: два объекта с одинаковым хешем не всегда оказы-

ваются одинаковыми. Это допускается, и любая реализация Python должна позволять исправлять такие конфликты. В CPython возможно *открытое решение* этой проблемы. Вероятность конфликта сильно влияет на производительность словаря, и если она высока, то словарь не получает бонусов к производительности от внутренней оптимизации.

Хотя три основные операции, такие как добавление, получение и удаление элемента, имеют среднюю сложность $O(1)$, их амортизированная сложность в худшем случае будет намного выше. Она сводится к $O(n)$, где n — текущий размер словаря. Кроме того, если в качестве ключей словаря служат пользовательские объекты класса и они хешируются неправильно (с высоким риском коллизий), то это окажет огромное негативное влияние на производительность словаря. Временные сложности CPython для словарей приведены в табл. 3.2.

Таблица 3.2

Операция	Средняя сложность	Амортизированная сложность в худшем случае
Получение элемента	$O(1)$	$O(n)$
Задание элемента	$O(1)$	$O(n)$
Удаление	$O(1)$	$O(n)$
Копирование	$O(n)$	$O(n)$
Перебор	$O(n)$	$O(n)$

Важно также знать, что число n в худшем случае сложности для копирования и перебора словаря — это максимальный размер, которого словарь когда-либо достигал, а не текущий размер. Иными словами, перебор словаря, когда-то огромного, а затем значительно сокращенного, будет выполняться невероятно долго. В отдельных случаях даже разумнее создать новый объект словаря, который будет гораздо меньше и обрабатываться будет быстрее.

Слабые стороны и альтернативные решения. В течение длительного времени одна из самых распространенных ошибок в словарях заключалась в том, что в них сохранялся порядок элементов, в которых добавлялись новые ключи. В Python 3.6 ситуация немного изменилась, а в Python 3.7 проблема была решена на уровне спецификации языка.

Но прежде, чем углубляться в Python 3.6 и более поздние версии, нам нужно слегка уйти от темы и исследовать проблему так, как если бы мы все еще застряли в прошлом, когда Python 3.6 еще не существовало. Раньше была возможна ситуация, когда последовательные ключи словаря имели последовательные хеши. В течение очень долгого времени это была единственная ситуация, в которой элементы словаря перебирались в том же порядке, в каком добавлялись в словарь. Самый